N-version Return Oriented Programming

One chain to rule them all

Arvind(dnivra)

SEFCOM
Arizona State University

May 5, 2019

$ whoami

- PhD student at SEFCOM, ASU
- Interested in reverse engineering and binary analysis.
- CTFing for a decade: playing, organizing and running a team.
- Also taught in university for a few years.

Code reuse attacks

- Most commonly used exploit technique.
- Circumvents every known defence technique practically implemented.
- Uses gadgets present in program code.
- “Write once, exploit everywhere.”

Diversity as a defence

- Comparable to genetic diversity to an extent.
- Multiple functionally equivalent copies(“variants”) with different instructions.
- Gadgets no longer universal.
- Eg: Gentoo Linux, Android.

Problem statement

Is software diversity an effective defence against code reuse attacks?

Contributions

- Twin gadgets: Valid gadgets present in different variants.
- Fully automated twin gadget extraction and N version ROP chain generation.
- Found average of 60 - 400 twin gadgets in 3 variants of 4 applications.
- $>$ 70 % converted to N version ROP chains for all variants.

Outline

Outline

Software diversity

- Software monoculture $\Longrightarrow$ “write once, run everywhere” exploits.
- Eliminate universal ROP gadgets using multiple non-identical copies.
- Compiler generated, different features, different software versions.
- “SoK: Automated Software Diversity”, IEEE S&P 2014.

Outline

Relooking at software diversity

- Eliminates identical gadgets from all binaries.
- But doesn't eliminate gadgets entirely: valid ROP gadgets still exist in all copies.
- Can be used to mount a code reuse attack.

Twin gadgets

- Gadgets at same address or offset in different variants.
- Twin gadget at 0x804b768 in nginx 1.3.9 compiled using clang, gcc and icc.

clang	gcc	icc
sbb al, 0x5e pop edi pop ebx pop ebp ret	ret	add [ebx - 0x75], ebx ret

Twin gadgets: some stats

Application	N = 2	N = 3	N = 4
bash	2838	184	17
nginx	3799	820	78
wireshark	2327	94	8
postgres	12077	879	42

Twin gadgets in N software versions compiled on Debian 8.

Outline

Finding gadgets

- Byte-by-byte search for gadgets using Capstone.
- Lift them to LLVM IR using BAP.
- Extract dependencies between registers in gadget for chaining them together.

Extracting dependencies

- Gadgets modify different registers and memory locations.
- Sequence of operations on multiple operands.
- Objective: Determine how operands depend on each other.
- Obtained by analysing LLVM IR but can be done using any IR I think.

Example dependency

- sbb al, 0x5e
  - EAX: 94, CF, EAX
- pop edi
  - EDI: Stack offset 0, size 4
- pop ebx
  - EBX: Stack offset 4, size 4
- pop ebp
  - EBP: Stack offset 8, size 4
- ret
  - ESP: ESP + 12

Outline

N version chain

- Control passes to start of N version chain.
- N version chain diverges execution of all variants.
- Regular ROP on each variant after divergence.
- Assumption: all variants start executing at same offset.

Generating chains

- Preprocessing: We focus on gadgets ending in *ret* without operands but sufficient to eliminate ones with privileged instructions.
- Identifying twin gadgets: Intersection of set of gadgets.
- Generate chains using twin gadgets found.

Generating chains(cont.)

- Case 1: Execution diverges for all variants.
- Variants: nginx 1.3.9 compiled using clang, gcc and icc.

clang	gcc	icc

pop esi
ret
	mov [ebx + 0xc4], eax
	mov eax, 0
	add esp, 0x18
	pop ebx
	ret
		pop ebx
		pop edi
		pop esi
		ret

Generating chains(cont.)

- Case 2: Execution diverges for some variants of complete set, V.
- Partition V using stack offset of return address into $\{V_1, V_2, ..., V_n\}$.
- Compute ROP chains, R_i for each V_i.
- Merge all R_i into N version chain R.

Chain generation walkthrough

Current N version chain

0	4	8	12	16
all	*gcc*, *icc*	0x41414141	0x41414141	*clang*

clang	gcc	icc
sbb al, 0x5e	ret	add [ebx - 0x75], ebx
pop edi		ret
pop ebx		
pop ebp		
ret		

Chain generation walkthrough

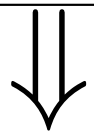
- Generate N version chain for gcc and icc.
- gcc: ret
- icc: pop ebp ; pop ebx ; pop esi ; ret

0	4	8	12	16
gcc, *icc*	*gcc*	0x41414141	0x41414141	*icc*

Chain generation walkthrough

0	4	8	12	16
all	*gcc*, *icc*	0x41414141	0x41414141	*clang*

0	4	8	12	16
gcc, *icc*	*gcc*	0x41414141	0x41414141	*icc*



0	4	8	12	16	20
all	*gcc*, *icc*	*gcc*	0x41414141	clang	*icc*

Generating chains(cont.)

- Case 3: Execution doesn't diverge for any variant in V.
- Could be useful for controlling some registers and hence, finding a chain.
- But potentially can become non-terminating.
- We don't explore such twin gadgets and discard them.

Challenges: finding successors

- Finding successor gadget a key challenge.
- Use dependency information to determine successor gadgets and select any one of them.
- Condition: Set of registers used in memory references should be a subset of registers controlled by predecessor.

Challenges: merge conflicts

- **Gadget-gadget**: Multiple chains want a gadget at same location in chain. Treated as not resolvable.
- **Gadget-memory**: Gadget on one and a memory address on another. If memory write, not resolvable.
- **Memory-memory**: If memory writes, not resolvable unless writeable location on all variants known.

Outline

Questions

- How many twin gadgets can be found?
- How many can be converted into complete chains?

Targets

Applications

- bash
- nginx
- postgres
- wireshark

Diversities

- Versions.
- Compiler optimization levels.
- Features.
- Compilers.
- Operating systems.

Twin gadget statistics

Twin gadget counts for 5 diversities with 3 variants

Application	μ	σ
bash	127	64
nginx	256	266
wireshark	67	60
postgres	409	206

ROP Chain statistics

N version ROP chain counts for 5 diversities with 3 variants

Application	μ	σ	Ratio
bash	114	51	89%
nginx	190	188	74%
wireshark	62	56	92%
postgres	311*	158*	94%*

*4 diversities.

Observations

- Highest twin gadget count: 1 version with different features.
- Lowest twin gadget count: No clear winner.
- $>$ 70 % of twin gadgets can be converted into an N version chain.

Demo

Demo and walkthrough of exploit for nginx 1.3.9 compiled using clang, gcc and icc using CVE-2013-2028.

Other targets

- About 100 Linux kernels diversified with a memory leak.
- Reasonable success but abandoned since it took too long.
- More possible ideas: combining different diversity techniques together, using CFI and much more.
- Increases difficulty but I think would still be possible.

Outline

Conclusion

- Software diversity not necessarily 100 % effective against ROP.
- Valid twin gadgets can form ROP chains which compromise a set of variants.
- Successfully generated N version exploits with different diversities.
- Need for better quantifying amount of diversity to determine effectiveness against ROP.