Efficient JOP Gadget Search

Quickstart: `cargo install xgadget --features cli-bin`

Google's 2022 analysis[1] of zero-day exploits "detected and disclosed as used in-the-wild" stated:

> *"Memory corruption vulnerabilities have been the standard for attacking software for the last few **decades** and it's **still** how attackers are having success."*

One factor in such incredible longevity is nascent adoption of memory-safe systems languages[2]. Another is continued emergence of new attack paradigms and techniques. Hardware $W \oplus X$ support (aka NX, DEP) has prevented *code injection* since the early 2000s. In response, ***Return Oriented Programming (ROP)*** introduced *code reuse*: an attacker with stack control chains together short, existing sequences of assembly (aka "gadgets") — should a leak enable computing gadget addresses in the face of ASLR. When contiguous ROP gadget addresses are written to a corrupted stack, each gadget's ending `ret` instruction pops the next gadget's address into the CPU's instruction pointer. The result? Turing-complete control over a victim process.

Jump Oriented Programming (JOP) is a newer code reuse method which, unlike ROP, doesn't rely on stack control. And thus bypasses shadow-stack implementations, like Intel CET SS[3]. JOP allows storing a *table* of gadget addresses in *any* RW memory location[4]. Instead of piggy-backing on call-return semantics to execute the gadget list, a "dispatch" gadget (e.g. `add rax, 8; jmp [rax]`) controls table indexing. Chaining happens if each gadget ends with a `jmp` back to the dispatcher (instead of a `ret`).

The Challenge in JOP Gadget Search

Disassembly is typically linear (decode consecutive instructions) or recursive-descent (follow control-flow from entry point). Gadget search is atypical: assuming x64, the ROP goal is finding every instance of an opcode (e.g. `0xc3`, 1 of 4 `ret` variants) and iteratively moving the disassembly starting point backwards, one byte at a time, to find a sequence of valid instructions ending with the tail opcode. Even if they start at misaligned offsets in the context of a normal program (e.g. partway through an intended instruction).

JOP gadgets present a unique challenge. For x64, the subset of relevant `jmp` and `call` instructions (e.g. `jmp rax` or `call [rbx]`, absolute indirect target) all have encodings starting with byte literal `0xff`. Most gadget search tools use regex to find *specific encodings* before attempting disassembly. For example, *certain* 4-byte encodings of `jmp [reg + offset]` match via `\xff[\x60-\x63\x65-\x67][\x00-\xff]`. Regex has two major drawbacks:

1. **Performance** — Must run the regex state machine to find matching offsets, then run a disassembler on matches (duplication of per-regex work).
2. **Completeness** — Need a complete list of regexs to match all 50+ possible x64 indirect `jmp`/`call` encodings (complex, error-prone).

Leveraging Instruction Semantics

We avoid both drawbacks with a general solution: encoding higher-level *operand semantics*. Attempt to disassemble a single instruction at every offset (or only instances of `0xff`), then work backwards if disassembly succeeds (e.g. valid instruction) and the instruction's operand *behavior* makes it a viable gadget tail.

The below code snippet finds JOP gadget tails, for all possible `jmp` and `call` encodings, using official Rust bindings for `zydis`[5].

```rust
#![no_std] // PROOF: below code is bare-metal portable
#![forbid(unsafe_code)] // PROOF: non-ext-lib code is mem-safe

use zydis::enums::{Mnemonic, OperandAction, OperandType};
use zydis::{DecodedInstruction, Register};

// Categorization ---------------------------------------

/// Check if viable JOP or COP tail instruction
pub fn is_jop_tail(instr: &DecodedInstruction) -> bool {
    matches!(instr.mnemonic, Mnemonic::JMP | Mnemonic::CALL)
        && (has_one_reg_op(instr) || has_one_reg_deref_op(instr))
}

// Constructs for attacker control ----------------------

/// Check for sole register operand (e.g. ``jmp rax'')
fn has_one_reg_op(instr: &DecodedInstruction) -> bool {
    instr
        .operands
        .iter()
        .filter(|&o| {
            (o.action == OperandAction::READ)
                && (o.ty == OperandType::REGISTER)
        }).count() == 1
}

/// Check for sole register-controlled memory
/// deference (e.g. ``jmp dword ptr [rax]'')
fn has_one_reg_deref_op(instr: &DecodedInstruction) -> bool {
    instr
        .operands
        .iter()
        .filter(|&o| {
            (o.action == OperandAction::READ)
                && (o.ty == OperandType::MEMORY)
                && (o.mem.base != Register::NONE)
        }).count() == 1
}
```

Closing

Society is still playing one of computer security's oldest cat-and-mouse games. If future exploit mitigations thwart ROP, JOP provides comparable expressivity — despite more complex gadget search and exploit development[6]. At least until safer type systems, CFI runtimes, and/or CHERI hardware become universal.

We've implemented the semantic search technique described here in `xgadget`[7] - a fast, parallel, open-source, *cross-{patch,compiler}-variant* ROP/JOP gadget finder. Happy hunting.

[1] https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html

[2] https://highassurance.rs

[3] Weakness: CET can include IBT to mitigate JOP. But IBT only validates target addrs, not func prototypes. Can still jump to imports, etc. JOP attacks are constrained, not eliminated.

[4] Aside: ROP chains may control stack location via "stack pivoting", but gadget address placement remains stack-restricted.

[5] https://zydis.re

[6] https://www.exploit-db.com/exploits/45045

[7] https://github.com/entropic-security/xgadget