compilers HATE him: use this ONE WEIRD TRICK to hide a message in your x86 program!!

william woodruff

!!con 2021

steganography (“steg”)

the art of hiding data in data

steganos / στεγανός – “concealed”

graphi / γραφή – “writing”

cryptography’s less formal, less useful sibling

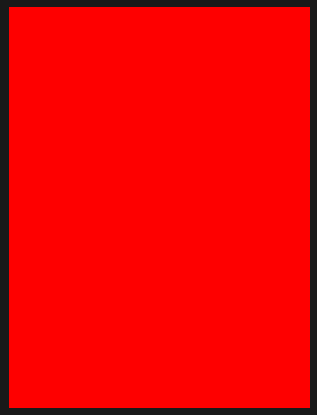
uses: covert channels, secret metadata, fun

theory of operation

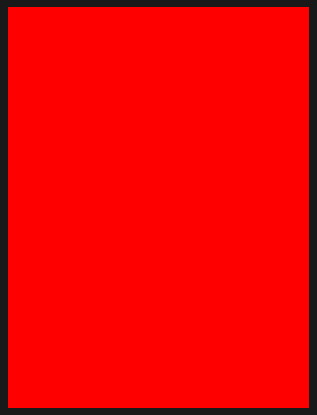
take a format whose contents can plausibly vary…

…and encode a secret message in those variations

RGB values can plausibly vary very slightly:



`#fe0000`



`#ff0000`

run-of-the-mill steg

images, video, audio, text files

×

```
steghide embed \
  -cf bigfoot.jpeg \
  -ef dbcooper.jpeg \
  -sf bigfootcooper.jpeg
```

```
steghide extract \
  -sf bigfootcooper.jpeg \
  -xf theperfectcrime.jpeg
```

run-of-the-mill steg

images, video, audio, text files

```
steghide embed \
  -cf bigfoot.jpeg \
  -ef dbcooper.jpeg \
  -sf bigfootcooper.jpeg
```

```
steghide extract \
  -sf bigfootcooper.jpeg \
  -xf theperfectcrime.jpeg
```

steg on computer programs

what would it look like to steg a compiled binary?

not like media formats:

- small changes produce significant behavioral changes
- transforms also have to preserve environmental assumptions
  - binary relocations, relative addressing, offsets, …

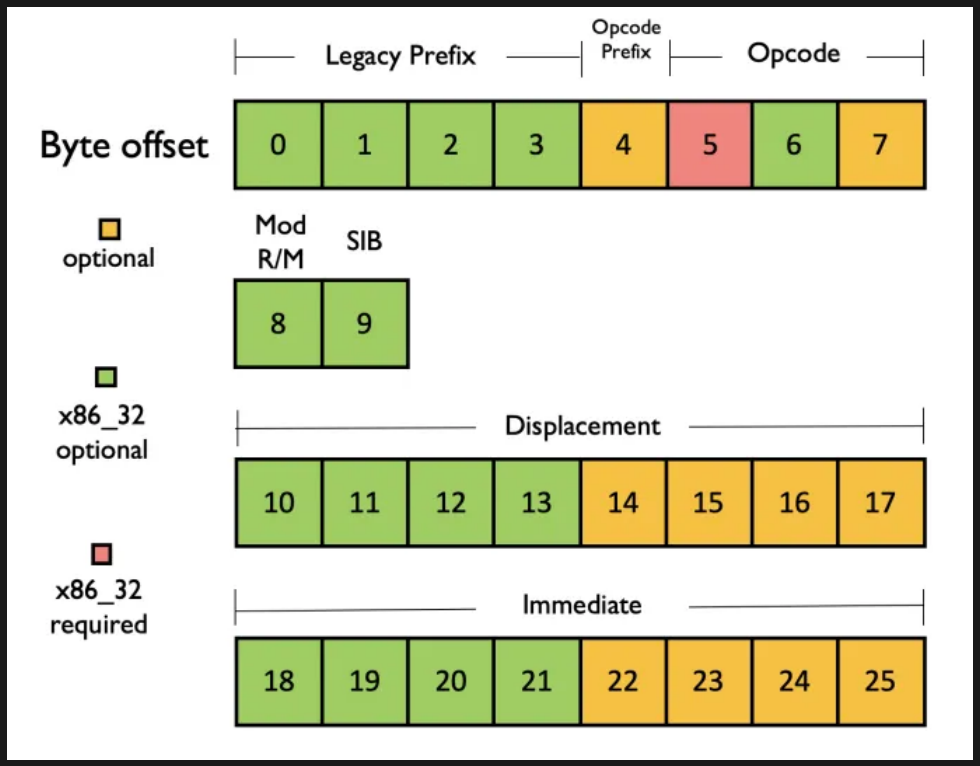
x86(-64)

- enough layers to give freud a nervous breakdown
  - 8008 (1972!) → 8080 (1974) → 8086 (1976) → 80386 (1985) → AMD64 (2003)
- CISC, variable length encoding (15 bytes max!)
- register-memory architecture

those last two make x86 *very* different from ARM, MIPS, RISC-V…

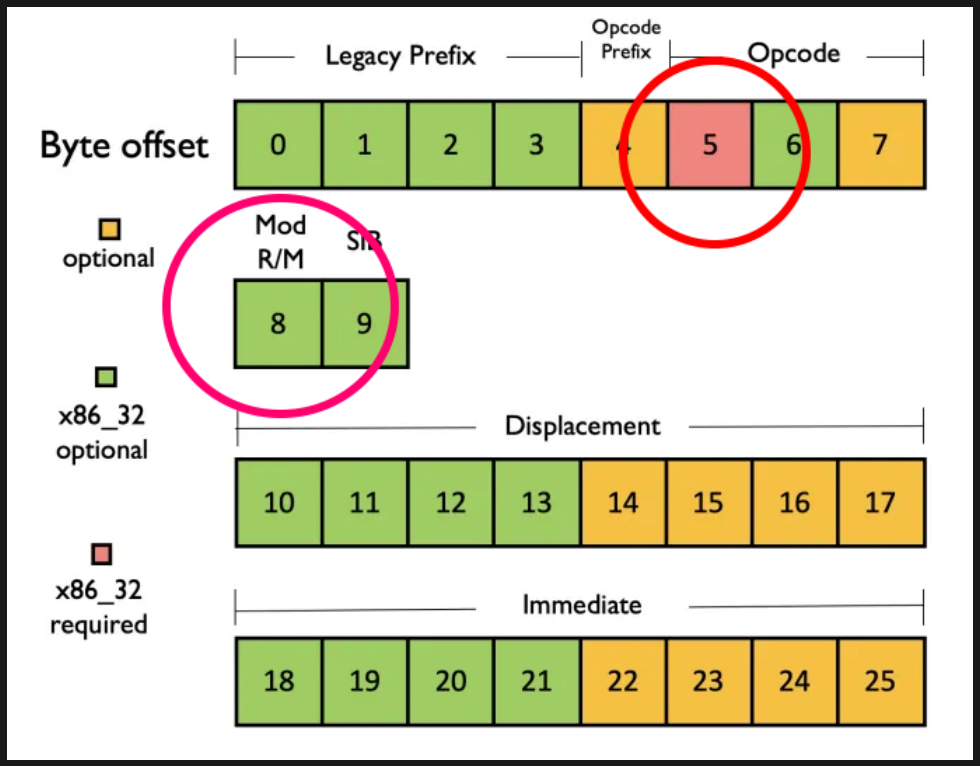
the x86 instruction format

(slightly overapproximated)



the x86 instruction format

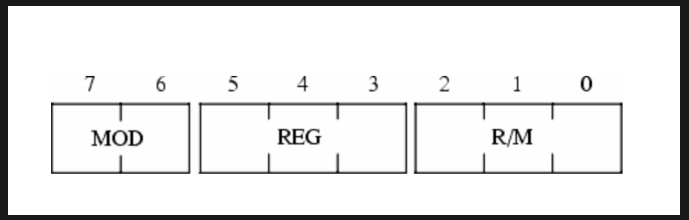
(slightly overapproximated)



operand encoding in x86

the ModR/M byte allows x86 to do these:

```
; operand order is controlled by OPC.d bit
ADD  eax, ebx   ; register-register
ADD  eax, [ebx] ; register-memory
ADD  [ebx], eax ; memory-register
```



- MOD – the “mode”
- REG – reg selector
- R/M – reg **or** memory

(AMD64 adds REX.R and REX.B for 64-bit regs)

duplicate encodings!!

`MODRM.REG` + `MODRM.RM` = 2 bits = 4 possible states

...but only three encoding forms 🤔

there are two ways to encoding register-register operands with ModR/M!!!!

duplicate encodings!!

Opcode	Mnemonic	Semantics
`01 /r`	`ADD r/m, r`	`r/m ← r/m + r`
`03 /r`	`ADD r, r/m`	`r ← r + r/m`

remember that `r/m` is either a register **or** memory operand!

duplicate encodings!!

Instruction	Encoding
`ADD eax, ebx`	`01h d8h`
`ADD eax, ebx`	`03h c3h`

- same length!
- identical semantics!!
- one bit of steganographic information per pair!!!

duplicate encodings!!

this trick works for a key subset of x86 instructions:
ADD, ADC, SUB, SBB, AND, OR, XOR, MOV, CMP

all of these are *extremely* common in x86 binaries!

works on 64-bit code too!

putting it all together

- secret message → secret bitstring
- target program → x86 instruction decoder
- iterate through decoded instructions
- rewrite them to match the secret bitstring!
- dump our new, fully functional executable

show me the code!

each supported instruction pair has 4 “variants” (8/16/32/64 bits)

not exactly how it works at the binary level, but that’s the disassembler’s view

```
static SEMANTIC_PAIRS: &[(Code, Code)] = &[
  // ADD
  (Code::Add_rm8_r8, Code::Add_r8_rm8),
  (Code::Add_rm16_r16, Code::Add_r16_rm16),
  (Code::Add_rm32_r32, Code::Add_r32_rm32),
  (Code::Add_rm64_r64, Code::Add_r64_rm64),
  // ... snip ...
];
```

show me the code!

we need to *profile* the program to see how much information we can store

for each disassembled instruction in the program:

```
// skip instructions we don't support
if !SUPPORTED_OPCODES.contains(&instruction.code()) {
    continue;
}

// skip non reg-to-reg variants
if instruction.op0_kind() != OpKind::Register
    || instruction.op1_kind() != OpKind::Register
{
    continue;
}

offsets.push(instruction.ip() as usize);
```

show me the code!

…store 1 bit of information by selecting from the pairs:

```rust
let new_code = {
    let tuple = SEMANTIC_PAIRS
        .iter()
        .find(|&&t| old_code == t.0 || old_code == t.1)
        .unwrap();

    match (bit, tuple.0 == old_code) {
        (false, true) | (true, false) => {
            // already correct!
            continue;
        }
        (false, false) => tuple.0,
        (true, true) => tuple.1,
    }
```

show me the code!

…and re-encode and rewrite the actual program:

```
let new_instruction = Instruction::with_reg_reg(
    new_code,
    instruction.op0_register(),
    instruction.op1_register(),
);
let new_len = encoder
    .encode(&new_instruction, offset as u64)
    .map_err(|s| anyhow!(s))?;

// ... snip ...

text_copy
    .data
    .splice(offset..(offset + new_len),
```

in action

give it a try: github.com/woodruffw/steg86

works on x86/AMD64 ELFs (Linux, BSD), PEs (Windows), Mach-Os (macOS), raw binaries

```
$ cargo install steg86
$ steg86 profile /bin/bash
$ steg86 embed /bin/bash /tmp/bash < top_secret
$ steg86 extract /tmp/bash > top_secret
```

thanks!

source: github.com/woodruffw/steg86

slides: yossarian.net/publications

blue bird site: @8x5clPW2