Object Diversification with the help of R2

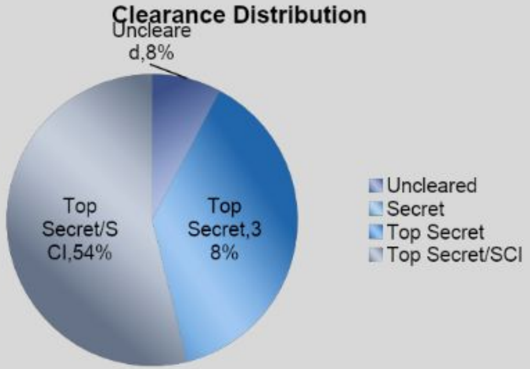
Alex Gaines, R2con 2019

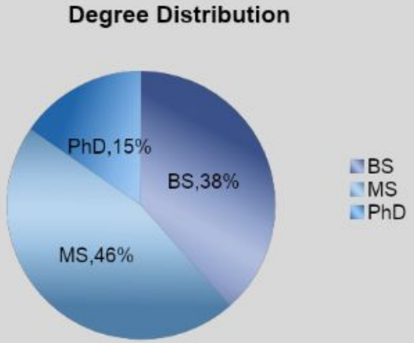
SIEGE technologies

This project is sponsored by the Intelligence Advanced Research Projects Activity (IARPA) under contract N66001-17-C-2003

Company Overview

- Founded 2009 by Jason Syversen
- 30+ Employees
- Most possess advanced degrees
- All have current (or in process for) clearances





About me:

Alex Gaines

Alex.Gaines@siegetechnologies.com

Siege Technologies.

I like bins.

A few R2 commits.

Also check out WPICTF.xyz

Automated Software Diversification - types

- Load Time
  - ASLR
    - On some embedded systems, may not be enabled or possible.
- Compile Time
  - Usually operate on AST or Source
  - Slow - must compile
- **<u>Binary</u>**
  - Can’t be very advanced.
  - Tends to break under certain cases.

Software Diversification. - Why

- Defense
  - Increase attacker workload.
    - Protect against Return Oriented Programming and other techniques.
  - Obfuscate.
  - Anti Piracy
- Offense
  - Avoid signatures.
    - AV/Intrusion detection avoidance.
  - Obfuscate

Goal

- Protect Kernel Modules from ROP attacks.
- Diversification
  - Every boot, Kernel Modules are different
  - Change ROP offsets, stack offsets/layout

Limitations

- Fast
- Small
- Reliable
- Operate on compiled kernel modules, not source

Quick Elf Overview

Executable and Linkable Format

Used often in Linux executables, objects, libraries, etc.

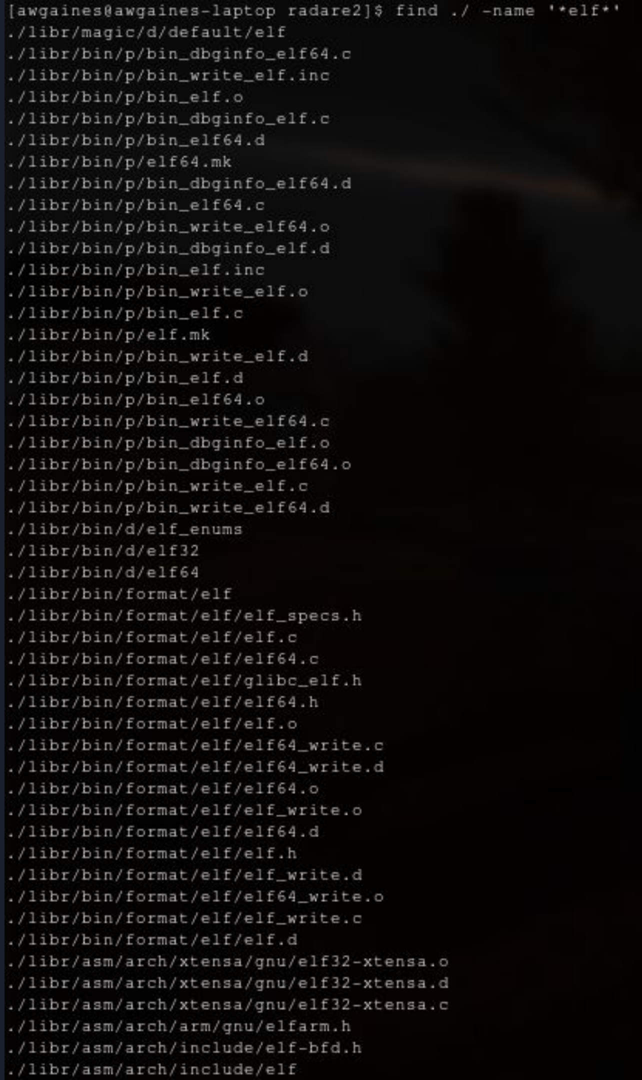
Consists of headers and sections.

We need to worry about Symbols and Relocations

R2 has elf code

But can also use other libraries, make your own, etc.

We made our own.



```
[awgaines@awgaines-laptop radare2]$ find ./ -name '*elf*'
./libr/magic/d/default/elf
./libr/bin/p/bin_dbginfo_elf64.c
./libr/bin/p/bin_write_elf.inc
./libr/bin/p/bin_elf.o
./libr/bin/p/bin_dbginfo_elf.c
./libr/bin/p/bin_elf64.d
./libr/bin/p/elf64.mk
./libr/bin/p/bin_dbginfo_elf64.d
./libr/bin/p/bin_elf64.c
./libr/bin/p/bin_write_elf64.o
./libr/bin/p/bin_dbginfo_elf.d
./libr/bin/p/bin_elf.inc
./libr/bin/p/bin_write_elf.o
./libr/bin/p/bin_elf.c
./libr/bin/p/elf.mk
./libr/bin/p/bin_write_elf.d
./libr/bin/p/bin_elf.d
./libr/bin/p/bin_elf64.o
./libr/bin/p/bin_write_elf64.c
./libr/bin/p/bin_dbginfo_elf.o
./libr/bin/p/bin_dbginfo_elf64.o
./libr/bin/p/bin_write_elf.c
./libr/bin/p/bin_write_elf64.d
./libr/bin/d/elf_enums
./libr/bin/d/elf32
./libr/bin/d/elf64
./libr/bin/format/elf
./libr/bin/format/elf/elf_specs.h
./libr/bin/format/elf/elf.c
./libr/bin/format/elf/elf64.c
./libr/bin/format/elf/glibc_elf.h
./libr/bin/format/elf/elf64.h
./libr/bin/format/elf/elf.o
./libr/bin/format/elf/elf64_write.c
./libr/bin/format/elf/elf64_write.d
./libr/bin/format/elf/elf64.o
./libr/bin/format/elf/elf_write.o
./libr/bin/format/elf/elf64.d
./libr/bin/format/elf/elf.h
./libr/bin/format/elf/elf_write.d
./libr/bin/format/elf/elf64_write.o
./libr/bin/format/elf/elf_write.c
./libr/bin/format/elf/elf.d
./libr/asm/arch/xtensa/gnu/elf32-xtensa.o
./libr/asm/arch/xtensa/gnu/elf32-xtensa.d
./libr/asm/arch/xtensa/gnu/elf32-xtensa.c
./libr/asm/arch/arm/gnu/elfarm.h
./libr/asm/arch/include/elf-bfd.h
./libr/asm/arch/include/elf
```

Easy Start

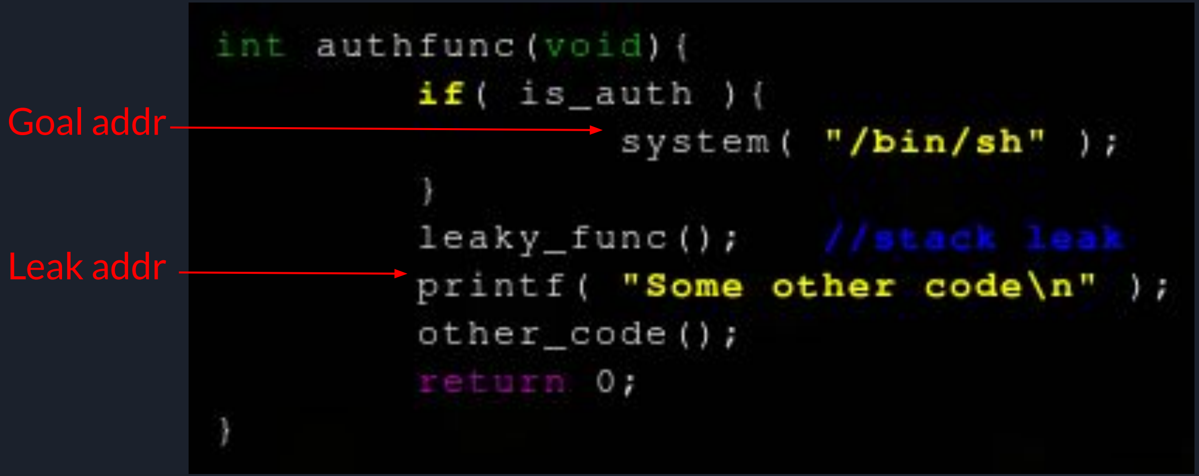
- Shuffle elf sections
    - -ffunction-sections splits every function into a section
    - Just update some ELF metadata indices.
    - OpenBSD does something similar
        - Re-Links LibC’s from .a files in a random order at every boot.
    - Limited # of permutations
        - More sections = better results

Easy Start

- Junk data at the end of sections
    - Random # of HLT instructions at the end of .text* sections
    - Comes after all the code
    - Changes offsets between sections.
        - But relative offsets inside sections are unchanged.

Why these aren't enough

- Only change offsets between sections, not inside sections.
- Limited permutations.
- Does not protect stack in any way.



```
int authfunc(void){
        if( is_auth ){
                system( "/bin/sh" );
        }
        leaky_func();    //stack leak
        printf( "Some other code\n" );
        other_code();
        return 0;
}
```

Let’s mess code up.

We need some help

- Change offsets between Leaked pointer and useful gadget.
    - Code insertion.
    - Can’t use “Trampoline style” dynamic code insertion. - SKORPIO.
- Want to modify the code itself.
    - Actually insert code, not just modify existing.
    - We need to “analyze” our sections.

Why Radare2?

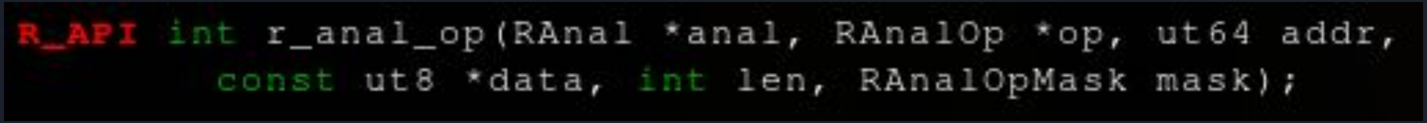
- Fast
- Low dependencies (No python required!)
- Compiles core to LibR
- Has some neat features (Esil)
- **I know how to use it**

Why libr and not r2pipe?

- **Fast**, a lot less overhead.
- Easier to use a real library than to interface with a text api.
  - At least for C.

Analysis overview.

Disassemble every executable section using r_anal_op() loops

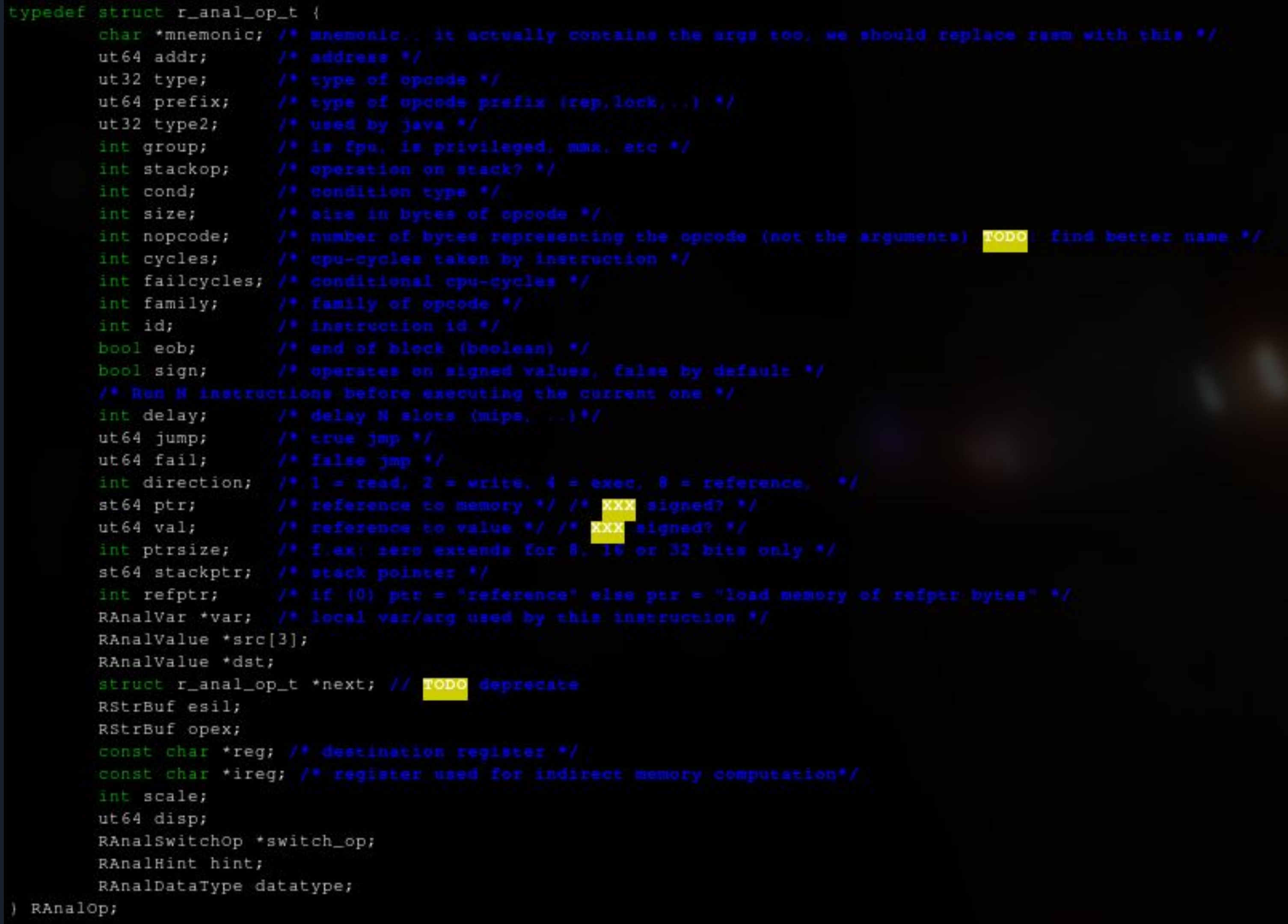


```
R_API int r_anal_op(RAnal *anal, RAnalOp *op, ut64 addr,
        const ut8 *data, int len, RAnalOpMask mask);
```

What this gives us:

- For each function (Symbol):
    - An ordered list of RAnalOps & their locations.
        - Instruction Type.
        - Jump targets.
        - Instruction opcode vs data size.
        - Registers used, memory access, etc.
        - **ESIL too!**

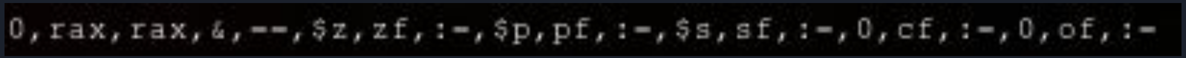
RAnalOp Is Super Useful.



```
typedef struct r_anal_op_t {
        char *mnemonic; /* mnemonic.. it actually contains the args too, we should replace rasm with this */
        ut64 addr;      /* address */
        ut32 type;      /* type of opcode */
        ut64 prefix;    /* type of opcode prefix (rep,lock,..) */
        ut32 type2;     /* used by java */
        int group;      /* is fpu, is privileged, mmx, etc */
        int stackop;    /* operation on stack? */
        int cond;       /* condition type */
        int size;       /* size in bytes of opcode */
        int nopcode;    /* number of bytes representing the opcode (not the arguments) TODO: find better name */
        int cycles;     /* cpu-cycles taken by instruction */
        int failcycles; /* conditional cpu-cycles */
        int family;     /* family of opcode */
        int id;         /* instruction id */
        bool eob;       /* end of block (boolean) */
        bool sign;      /* operates on signed values, false by default */
        /* Run N instructions before executing the current one */
        int delay;      /* delay N slots (mips, ..)*/
        ut64 jump;      /* true jmp */
        ut64 fail;      /* false jmp */
        int direction;  /* 1 = read, 2 = write, 4 = exec, 8 = reference,  */
        st64 ptr;       /* reference to memory */ /* XXX signed? */
        ut64 val;       /* reference to value */ /* XXX signed? */
        int ptrsize;    /* f.ex: zero extends for 8, 16 or 32 bits only */
        st64 stackptr;  /* stack pointer */
        int refptr;     /* if (0) ptr = "reference" else ptr = "load memory of refptr bytes" */
        RAnalVar *var;  /* local var/arg used by this instruction */
        RAnalValue *src[3];
        RAnalValue *dst;
        struct r_anal_op_t *next; // TODO deprecate
        RStrBuf esil;
        RStrBuf opex;
        const char *reg; /* destination register */
        const char *ireg; /* register used for indirect memory computation*/
        int scale;
        ut64 disp;
        RAnalSwitchOp *switch_op;
        RAnalHint hint;
        RAnalDataType datatype;
} RAnalOp;
```

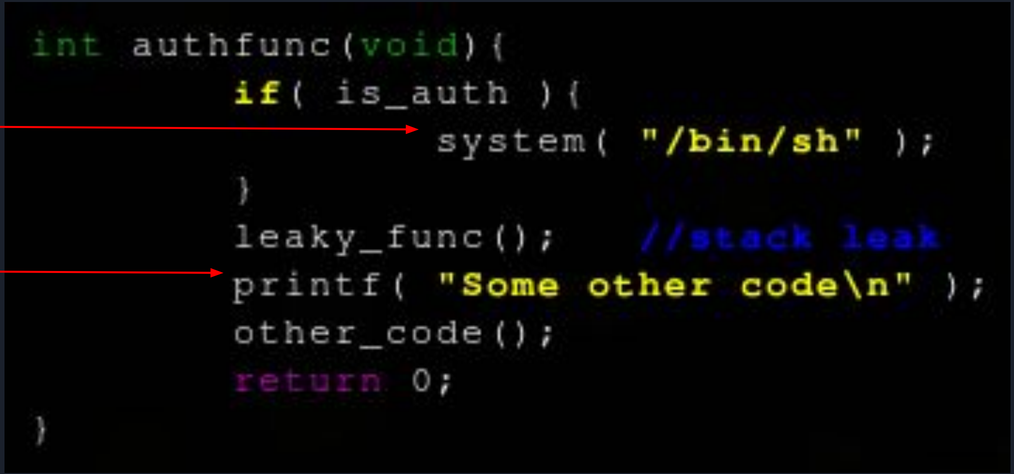
Esil is great too

- Great for easily gathering instruction information.
    - strstr() is your greatest friend here.
- Not terribly great at performance, but it’s useful to get something working.
    - Does this instruction use the stack? strstr(esil, “rsp”)



```
0,rax,rax,&,==,$z,zf,:=,$p,pf,:=,$s,sf,:=,0,cf,:=,0,of,:=
```

“Cruftables”

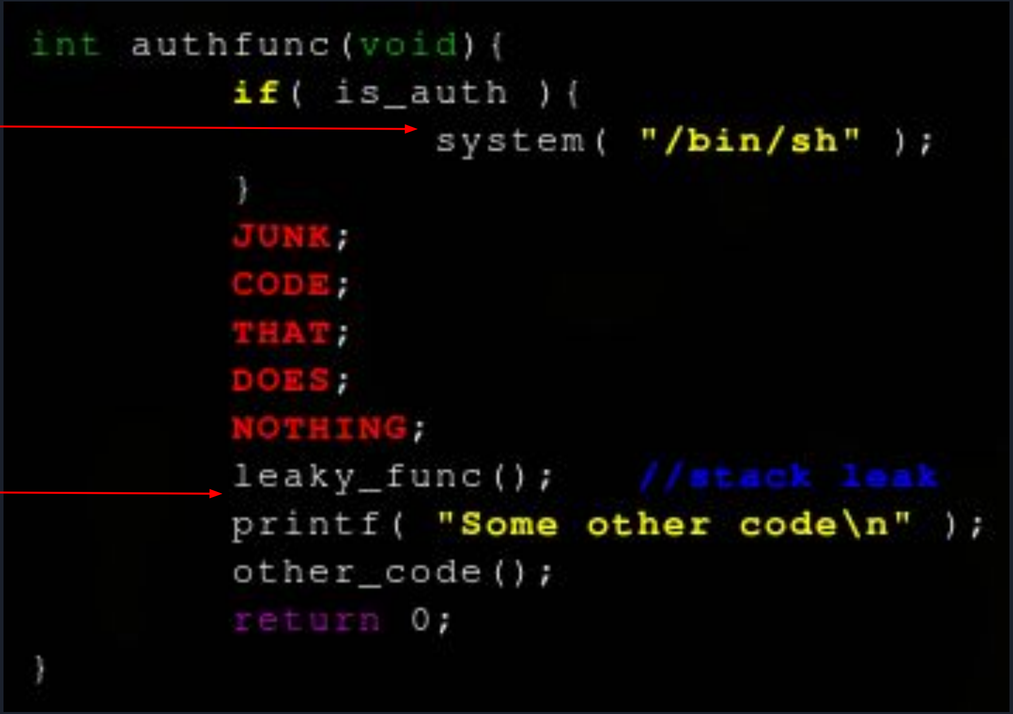


```
int authfunc(void){
        if( is_auth ){
                system( "/bin/sh" );
        }
        leaky_func();    //stack leak
        printf( "Some other code\n" );
        other_code();
        return 0;
}
```

Goal addr

Leak addr

“Cruftables”



```
int authfunc(void){
        if( is_auth ){
                system( "/bin/sh" );
        }
        JUNK;
        CODE;
        THAT;
        DOES;
        NOTHING;
        leaky_func();    //stack leak
        printf( "Some other code\n" );
        other_code();
        return 0;
}
```

Goal addr

Leak addr

What needs to be done?

Decide where to put a “cruftable”.

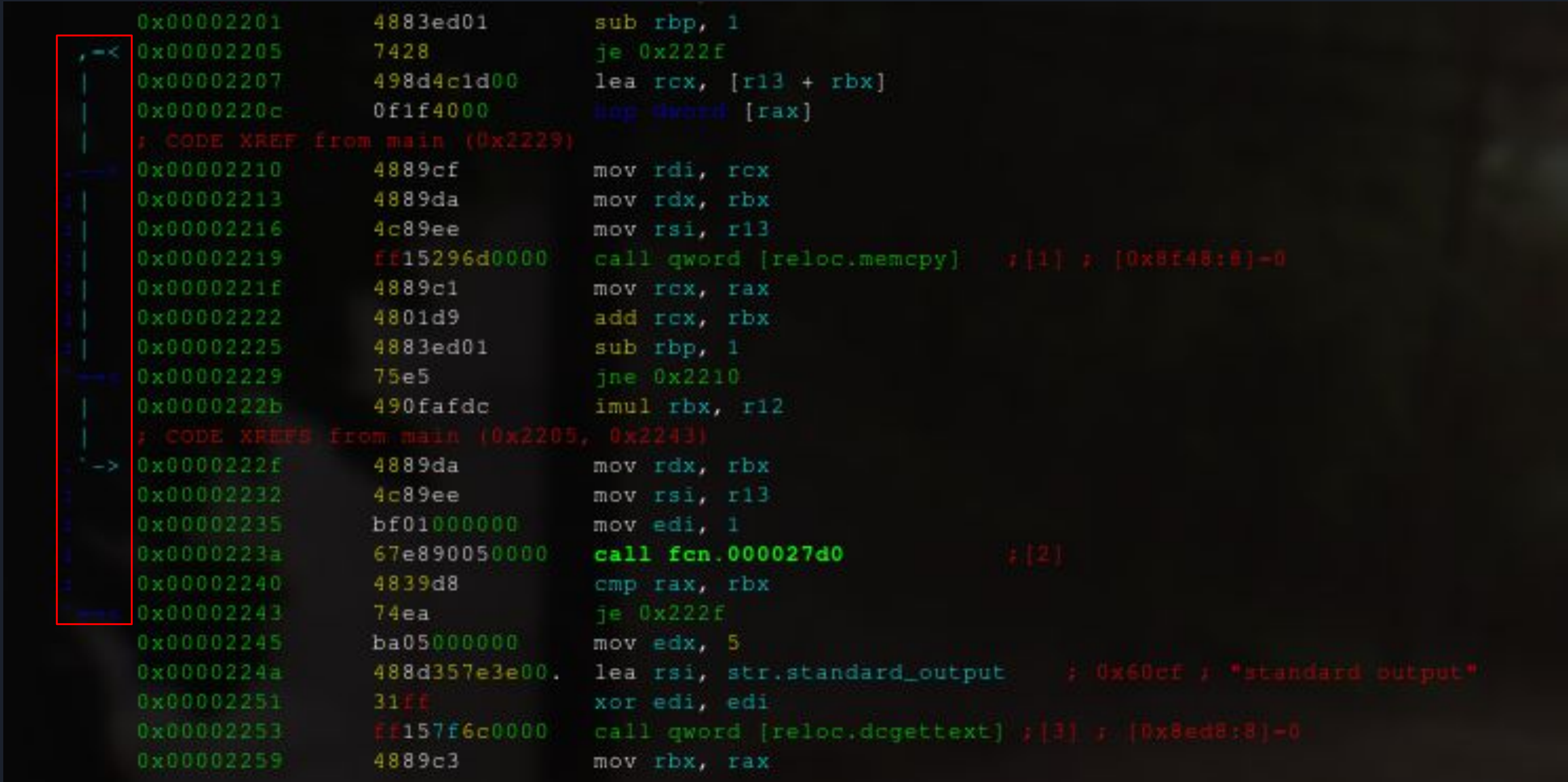
Decide what data is a “cruftable”?

Adjust symbols, relocs, etc to accommodate cruftable.

Where can we put a Cruftable?

- We have to insert in between instructions.
  - Can’t cut an instruction in half.
- Can’t put a cruftable in a “badzone”
  - Badzone consists of anywhere between a jump instruction and its jump target.
  - Not quite a basic block, but similar.

Badzone(s)

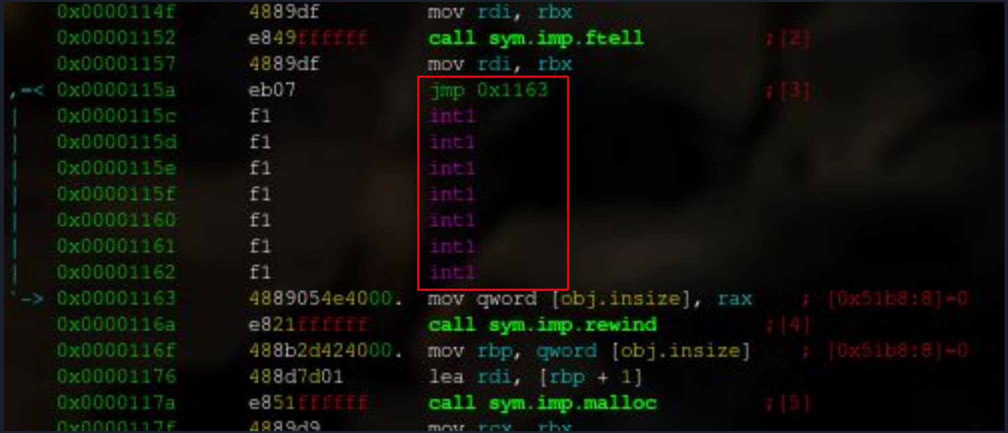


```
      0x00002201      4883ed01       sub rbp, 1
  ,=< 0x00002205      7428           je 0x222f
  |   0x00002207      498d4c1d00     lea rcx, [r13 + rbx]
  |   0x0000220c      0f1f4000       nop dword [rax]
  |   ; CODE XREF from main (0x2229)
 .--> 0x00002210      4889cf         mov rdi, rcx
 :|   0x00002213      4889da         mov rdx, rbx
 :|   0x00002216      4c89ee         mov rsi, r13
 :|   0x00002219      ff15296d0000   call qword [reloc.memcpy]   ;[1] ; [0x8f48:8]=0
 :|   0x0000221f      4889c1         mov rcx, rax
 :|   0x00002222      4801d9         add rcx, rbx
 :|   0x00002225      4883ed01       sub rbp, 1
 `==< 0x00002229      75e5           jne 0x2210
  |   0x0000222b      490fafdc       imul rbx, r12
  |   ; CODE XREFS from main (0x2205, 0x2243)
 .`-> 0x0000222f      4889da         mov rdx, rbx
 :    0x00002232      4c89ee         mov rsi, r13
 :    0x00002235      bf01000000     mov edi, 1
 :    0x0000223a      67e890050000   call fcn.000027d0           ;[2]
 :    0x00002240      4839d8         cmp rax, rbx
 `==< 0x00002243      74ea           je 0x222f
      0x00002245      ba05000000     mov edx, 5
      0x0000224a      488d357e3e00.  lea rsi, str.standard_output    ; 0x60cf ; "standard output"
      0x00002251      31ff           xor edi, edi
      0x00002253      ff157f6c0000   call qword [reloc.dcgettext] ;[3] ; [0x8ed8:8]=0
      0x00002259      4889c3         mov rbx, rax
```

RAnalOp to Badzone.

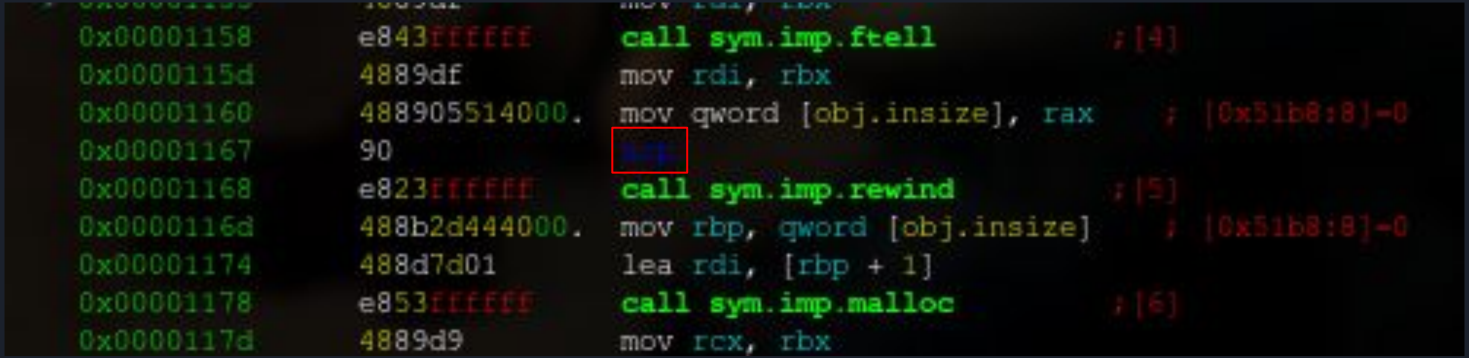
- RAnalOp has a lot of useful data to use.
- Many ways to determine badzone-causing Instruction.
  - **Most important (RAnalOp.jump != -1 || RAnalOp.fail != -1)**
- Badzone = Bounds (instr.loc, instr.loc + instr.size, RAnalOp.jump, RAnalOp.fail)

Data in a cruftable.



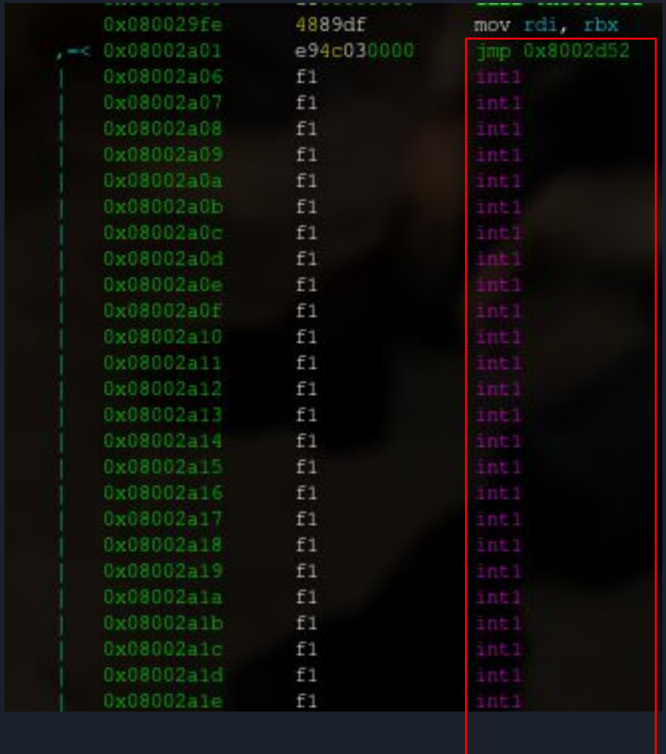
```
     0x0000114f      4889df          mov rdi, rbx
     0x00001152      e849ffffff      call sym.imp.ftell          ;[2]
     0x00001157      4889df          mov rdi, rbx
,=< 0x0000115a      eb07            jmp 0x1163                  ;[3]
|    0x0000115c      f1              int1
|    0x0000115d      f1              int1
|    0x0000115e      f1              int1
|    0x0000115f      f1              int1
|    0x00001160      f1              int1
|    0x00001161      f1              int1
|    0x00001162      f1              int1
`-> 0x00001163      4889054e4000.   mov qword [obj.insize], rax    ; [0x51b8:8]=0
     0x0000116a      e821ffffff      call sym.imp.rewind         ;[4]
     0x0000116f      488b2d424000.   mov rbp, qword [obj.insize]    ; [0x51b8:8]=0
     0x00001176      488d7d01        lea rdi, [rbp + 1]
     0x0000117a      e851ffffff      call sym.imp.malloc         ;[5]
     0x0000117f      4889d9          mov rcx, rbx
```

Data in a cruftable.



```
0x00001158      e843ffffff      call sym.imp.ftell          ;[4]
0x0000115d      4889df          mov rdi, rbx
0x00001160      488905514000.   mov qword [obj.insize], rax    ; [0x51b8:8]=0
0x00001167      90              nop
0x00001168      e823ffffff      call sym.imp.rewind         ;[5]
0x0000116d      488b2d444000.   mov rbp, qword [obj.insize]    ; [0x51b8:8]=0
0x00001174      488d7d01        lea rdi, [rbp + 1]
0x00001178      e853ffffff      call sym.imp.malloc         ;[6]
0x0000117d      4889d9          mov rcx, rbx
```

Cruftables can be long.

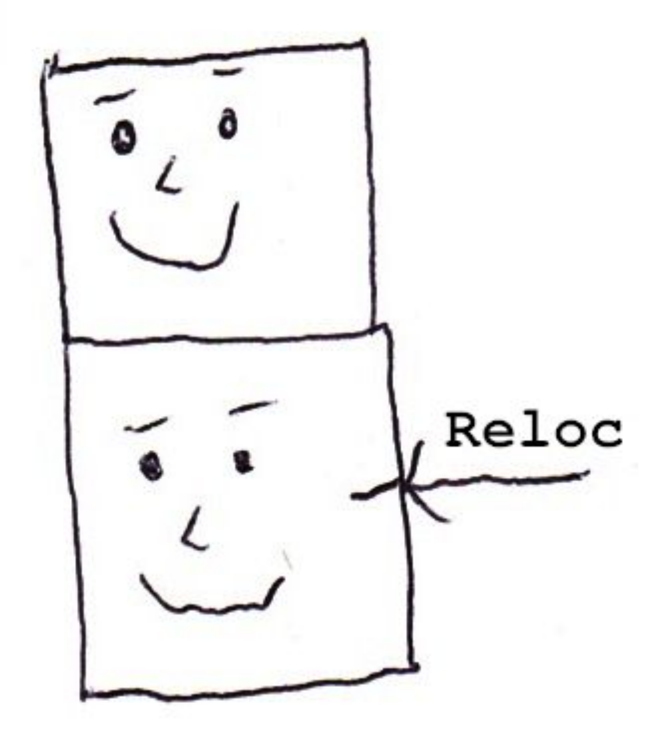


```
        0x080029fe      4889df        mov rdi, rbx
   ,=<  0x08002a01      e94c030000    jmp 0x8002d52
   |    0x08002a06      f1            int1
   |    0x08002a07      f1            int1
   |    0x08002a08      f1            int1
   |    0x08002a09      f1            int1
   |    0x08002a0a      f1            int1
   |    0x08002a0b      f1            int1
   |    0x08002a0c      f1            int1
   |    0x08002a0d      f1            int1
   |    0x08002a0e      f1            int1
   |    0x08002a0f      f1            int1
   |    0x08002a10      f1            int1
   |    0x08002a11      f1            int1
   |    0x08002a12      f1            int1
   |    0x08002a13      f1            int1
   |    0x08002a14      f1            int1
   |    0x08002a15      f1            int1
   |    0x08002a16      f1            int1
   |    0x08002a17      f1            int1
   |    0x08002a18      f1            int1
   |    0x08002a19      f1            int1
   |    0x08002a1a      f1            int1
   |    0x08002a1b      f1            int1
   |    0x08002a1c      f1            int1
   |    0x08002a1d      f1            int1
   |    0x08002a1e      f1            int1
```

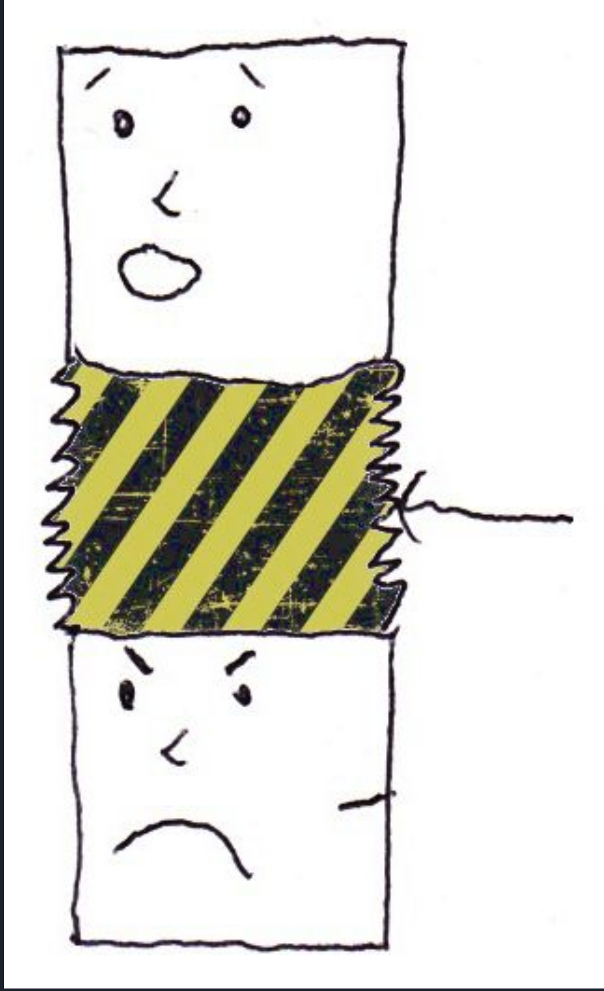
Elf data adjustment.

- Any Symbol that comes after a cruftable needs to be adjusted.
  - (For every Symbol: if Symbol is after cruftable, add cruftable size to Symbol)
- Any Reloc that comes after a cruftable needs to be adjusted.
  - (For every Reloc: if Reloc is after cruftable, add cruftable size to Reloc)

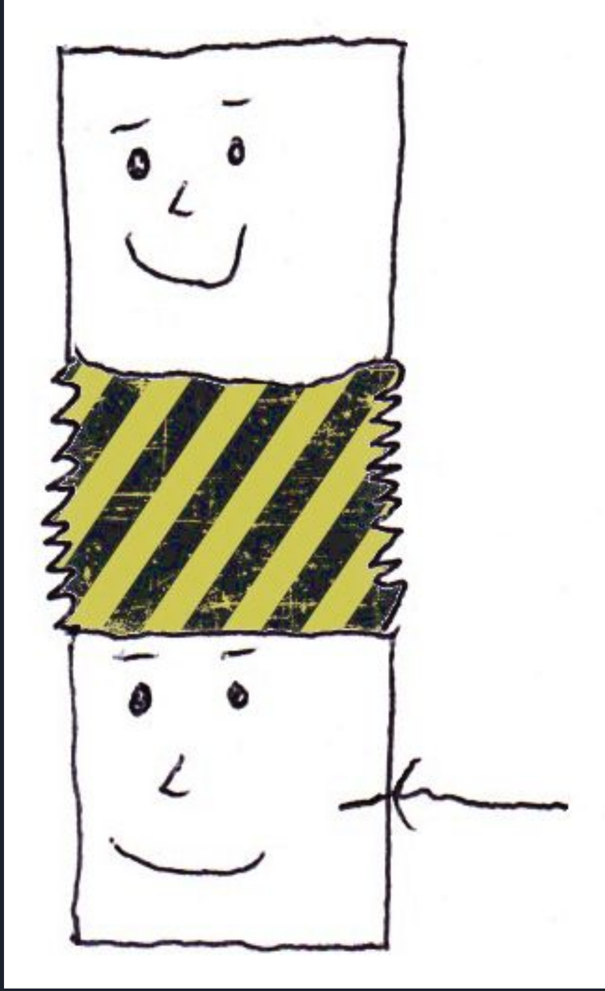
Elf data adjustment.



Elf data adjustment.

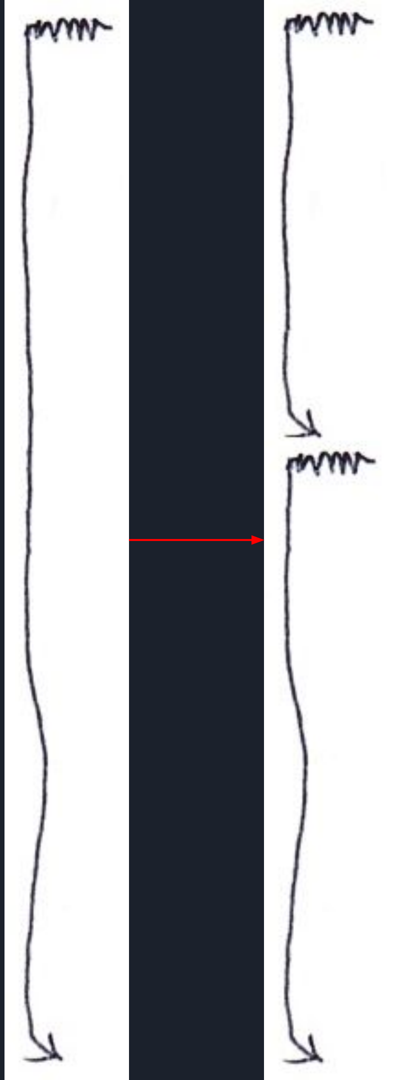


Elf data adjustment.

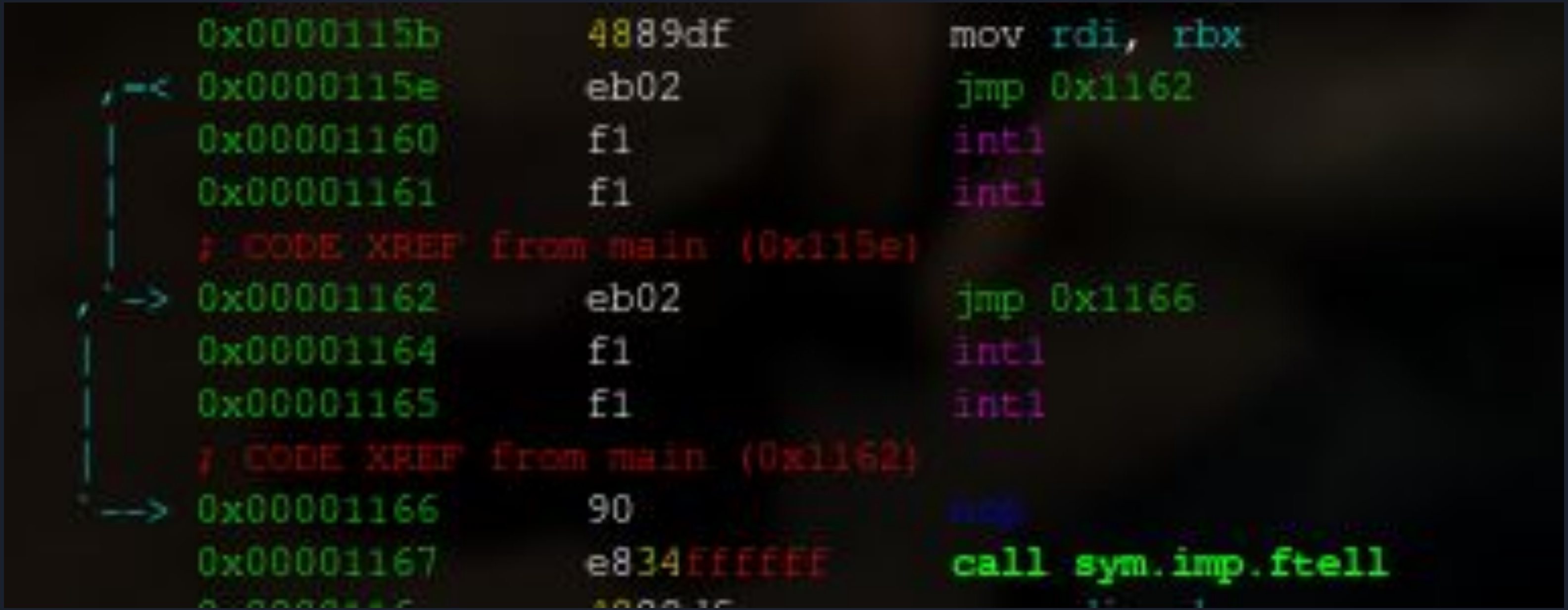


Split Cruftables

- When Inserting a cruftable: Randomly split into 2 cruftables.
    - Can be applied recursively.



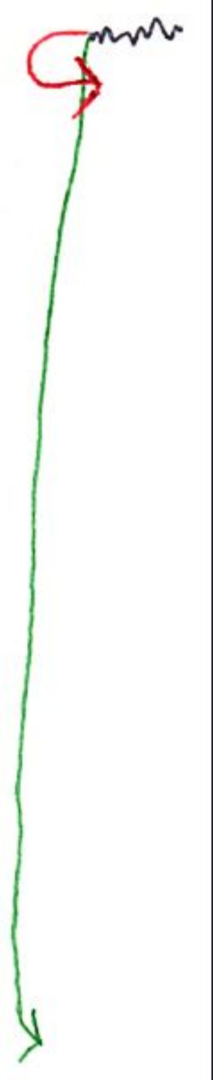
A split cruftable.



```
        0x0000115b      4889df          mov rdi, rbx
  ,=< 0x0000115e      eb02            jmp 0x1162
  |   0x00001160      f1              int1
  |   0x00001161      f1              int1
  |   ; CODE XREF from main (0x115e)
 ,`-> 0x00001162      eb02            jmp 0x1166
 |    0x00001164      f1              int1
 |    0x00001165      f1              int1
 |    ; CODE XREF from main (0x1162)
 `--> 0x00001166      90              nop
      0x00001167      e834ffffff      call sym.imp.ftell
```

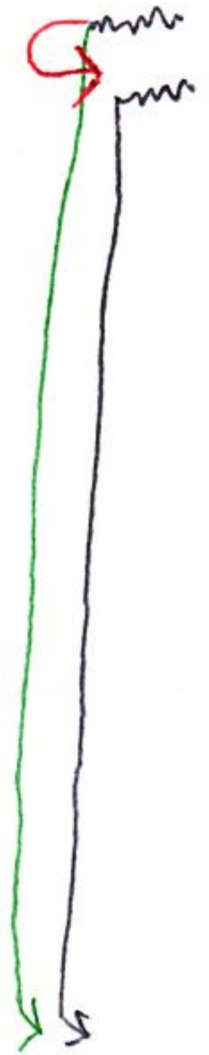
Confused Cruftables

- Instead of using jumps, use conditional jumps.
- “Fail” path must have a another cruftable inserted after it to make sure it correctly reaches the end.
  - Can be recursive, can be split.
  - Easily done recursively.
- When combined with split cruftables, it creates gnarly code.

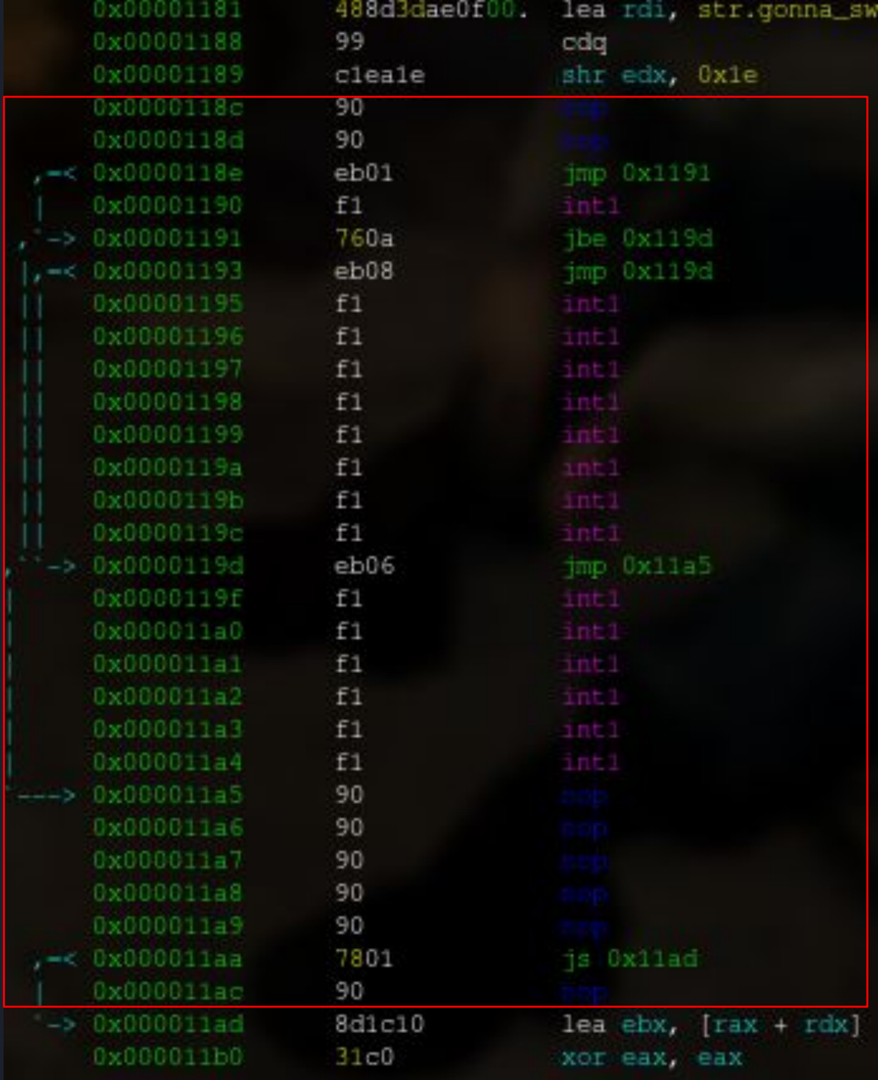


Confused Cruftables

- Instead of using jumps, use conditional jumps.
- “Fail” path must have a another cruftable inserted after it to make sure it correctly reaches the end.
  - Can be recursive, can be split.
  - Easily done recursively.
- When combined with split cruftables, it creates gnarly code.



Confused and split cruftable

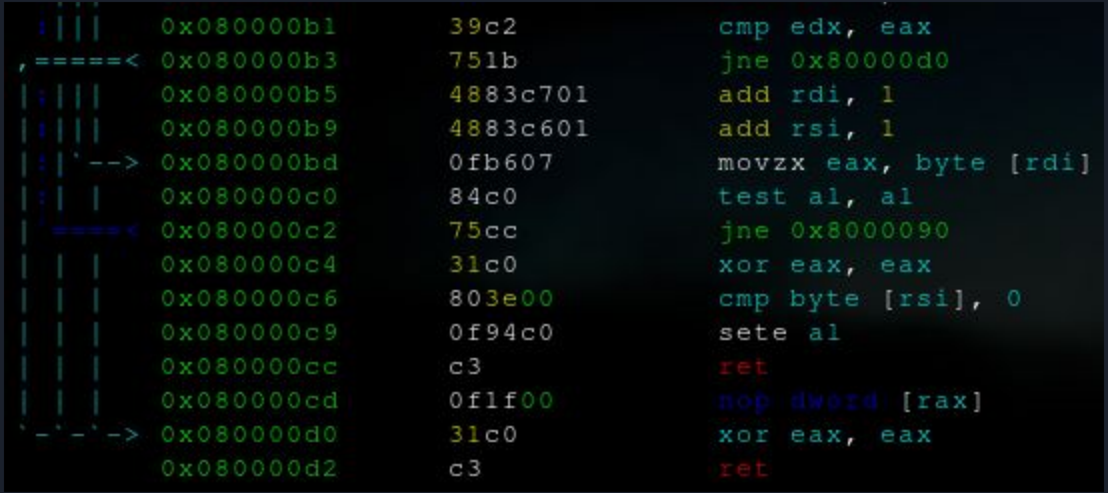


```
          0x00001181      488d3dae0f00.  lea rdi, str.gonna_sw
          0x00001188      99             cdq
          0x00001189      c1ea1e         shr edx, 0x1e
          0x0000118c      90             nop
          0x0000118d      90             nop
     ,=< 0x0000118e      eb01           jmp 0x1191
     |   0x00001190      f1             int1
    ,`-> 0x00001191      760a           jbe 0x119d
    |,=< 0x00001193      eb08           jmp 0x119d
    ||   0x00001195      f1             int1
    ||   0x00001196      f1             int1
    ||   0x00001197      f1             int1
    ||   0x00001198      f1             int1
    ||   0x00001199      f1             int1
    ||   0x0000119a      f1             int1
    ||   0x0000119b      f1             int1
    ||   0x0000119c      f1             int1
   ,``-> 0x0000119d      eb06           jmp 0x11a5
   |     0x0000119f      f1             int1
   |     0x000011a0      f1             int1
   |     0x000011a1      f1             int1
   |     0x000011a2      f1             int1
   |     0x000011a3      f1             int1
   |     0x000011a4      f1             int1
   `---> 0x000011a5      90             nop
         0x000011a6      90             nop
         0x000011a7      90             nop
         0x000011a8      90             nop
         0x000011a9      90             nop
     ,=< 0x000011aa      7801           js 0x11ad
     |   0x000011ac      90             nop
     `-> 0x000011ad      8d1c10         lea ebx, [rax + rdx]
         0x000011b0      31c0           xor eax, eax
```

Super Cruftables

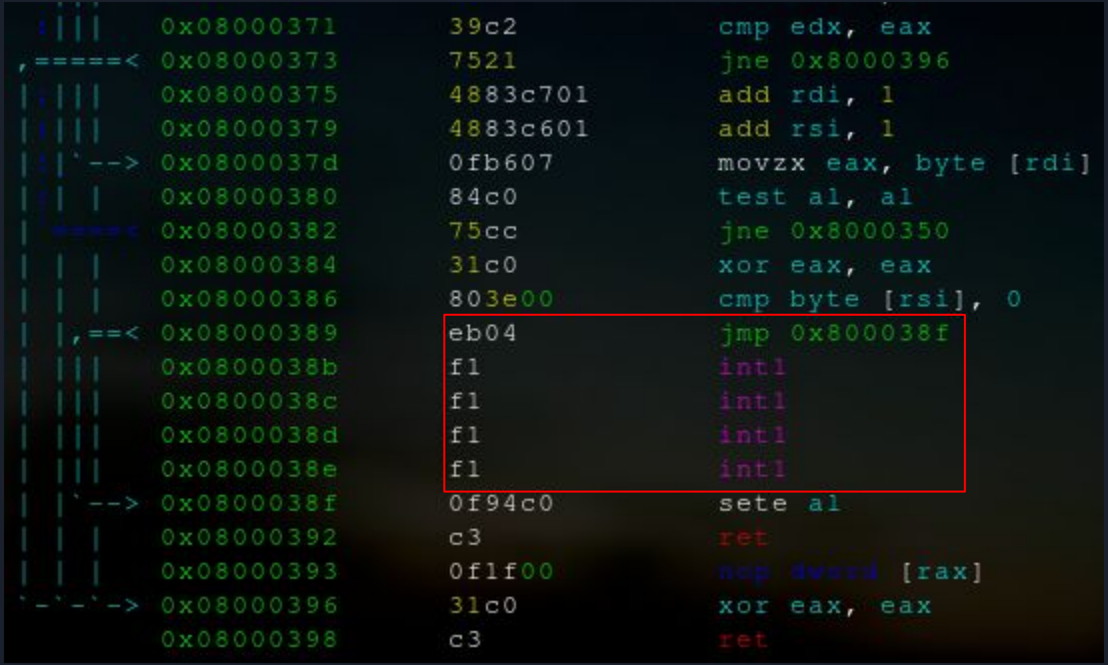
Cruftables that can be inserted anywhere, even inside badzones.

Super Cruftables



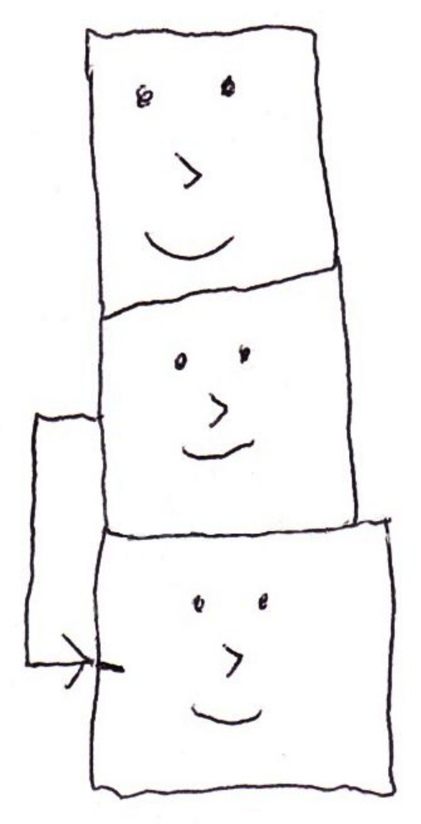
```
 :|||    0x080000b1    39c2        cmp edx, eax
,=====< 0x080000b3    751b        jne 0x80000d0
|:|||    0x080000b5    4883c701    add rdi, 1
|:|||    0x080000b9    4883c601    add rsi, 1
|:|`--> 0x080000bd    0fb607      movzx eax, byte [rdi]
|:| |    0x080000c0    84c0        test al, al
| ====< 0x080000c2    75cc        jne 0x8000090
| | |    0x080000c4    31c0        xor eax, eax
| | |    0x080000c6    803e00      cmp byte [rsi], 0
| | |    0x080000c9    0f94c0      sete al
| | |    0x080000cc    c3          ret
| | |    0x080000cd    0f1f00      nop dword [rax]
`-`-`-> 0x080000d0    31c0        xor eax, eax
         0x080000d2    c3          ret
```

Super Cruftables

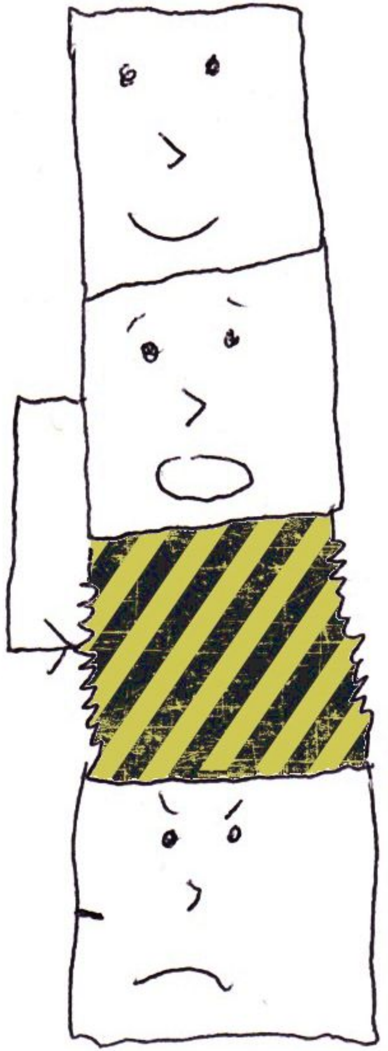


```
  :|||    0x08000371      39c2          cmp edx, eax
,=====<   0x08000373      7521          jne 0x8000396
|:|||     0x08000375      4883c701      add rdi, 1
|:|||     0x08000379      4883c601      add rsi, 1
|:|`-->   0x0800037d      0fb607        movzx eax, byte [rdi]
|:| |     0x08000380      84c0          test al, al
| =====<  0x08000382      75cc          jne 0x8000350
| | |     0x08000384      31c0          xor eax, eax
| | |     0x08000386      803e00        cmp byte [rsi], 0
| |,==<   0x08000389      eb04          jmp 0x800038f
| |||     0x0800038b      f1            int1
| |||     0x0800038c      f1            int1
| |||     0x0800038d      f1            int1
| |||     0x0800038e      f1            int1
| |`-->   0x0800038f      0f94c0        sete al
| | |     0x08000392      c3            ret
| | |     0x08000393      0f1f00        nop dword [rax]
`-`-`-->  0x08000396      31c0          xor eax, eax
          0x08000398      c3            ret
```

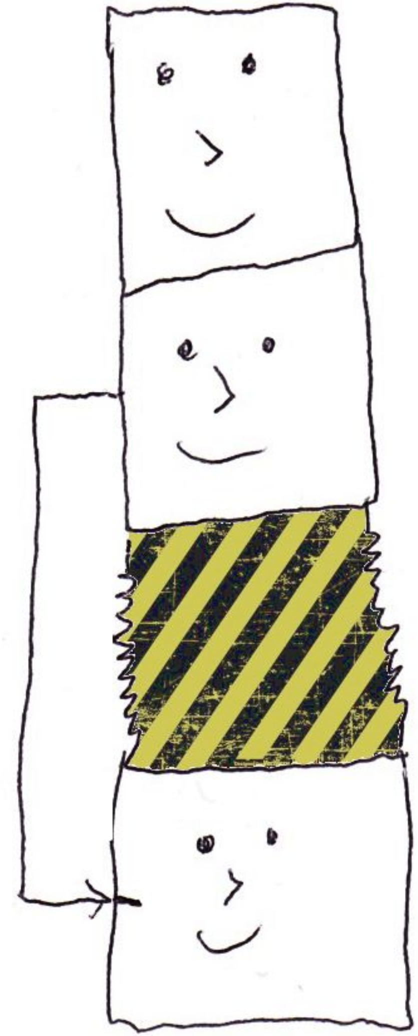
Just update jump offsets



Just update jump offsets

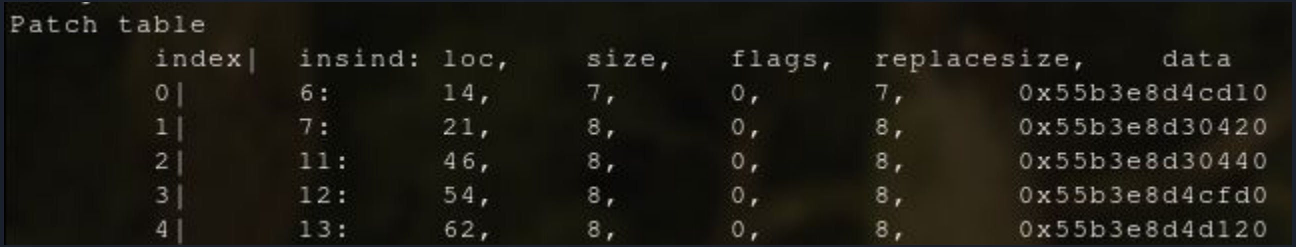


Just update jump offsets



Patch System

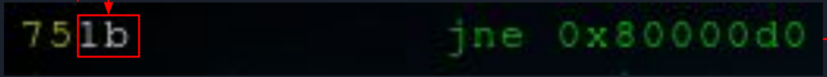
- Keep list of all wanted insertions/removals/changes.
  - Apply periodically.
- Attempt each one, ignore if impossible.
- Modifies Jump offsets, symbols, relocs, etc



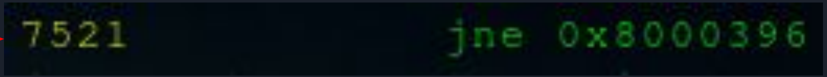
```
Patch table
        index|  insind: loc,    size,   flags,  replacesize,    data
        0|      6:      14,     7,      0,      7,      0x55b3e8d4cd10
        1|      7:      21,     8,      0,      8,      0x55b3e8d30420
        2|      11:     46,     8,      0,      8,      0x55b3e8d30440
        3|      12:     54,     8,      0,      8,      0x55b3e8d4cfd0
        4|      13:     62,     8,      0,      8,      0x55b3e8d4d120
```

Patch System modify jump offsets

- If cruftable is inserted inside a badzone, add cruftable size to jump offset.
  - Instruction start + RAnalOp->nopsize is the start of the offset
  - RAnalOp->size - RAnalOp->nopsize is the size of the offset
    - (int8_t, int16_t, int32_t, etc)



```
751b                    jne 0x80000d0
```



```
7521                    jne 0x8000396
```

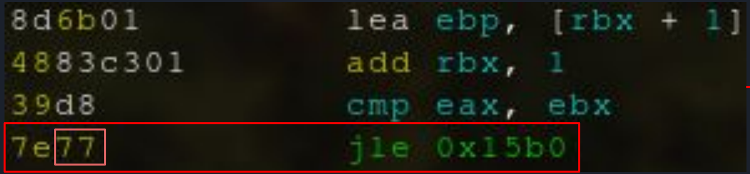
Instruction Bumping

Jumps are Short (1 byte signed) or Near (4 byte signed).

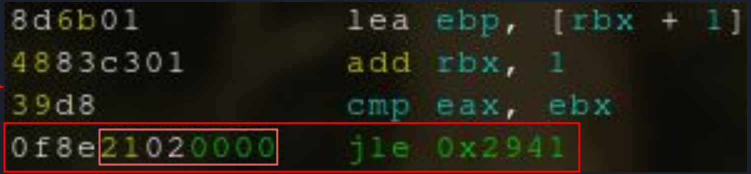
-128 -> 127 vs -2Gigs + 2Gigs

If we go over -128->127 limit, we must “bump” instruction to larger version.

Apply recursively



```
8d6b01          lea ebp, [rbx + 1]
4883c301        add rbx, 1
39d8            cmp eax, ebx
7e77            jle 0x15b0
```

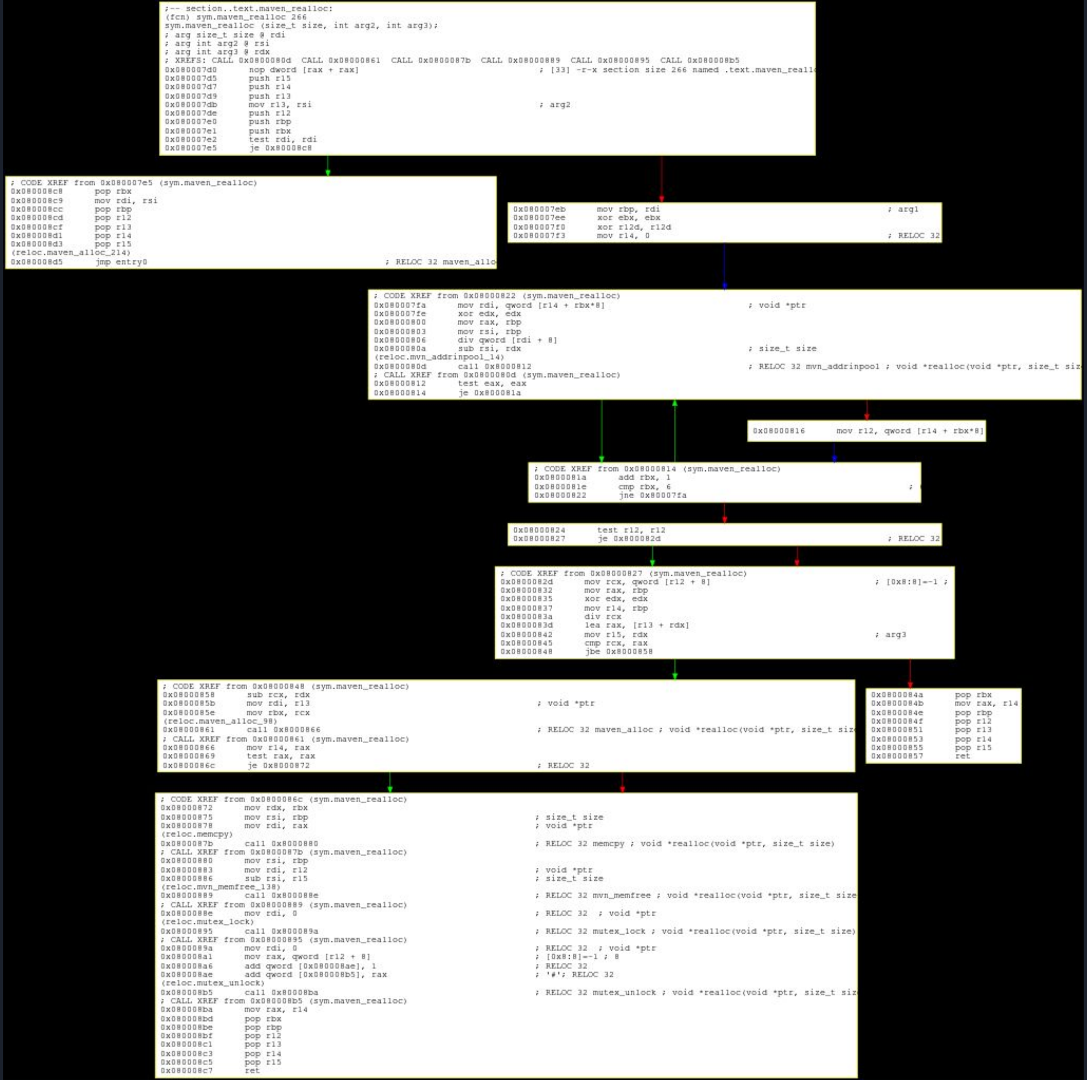


```
8d6b01          lea ebp, [rbx + 1]
4883c301        add rbx, 1
39d8            cmp eax, ebx
0f8e21020000    jle 0x2941
```

One can’t be bumped

- E3 - JECXZ
    - Only a short jump version, no near jump
- We have to ignore patches that would result in the bumping of JECXZ.
    - Thankfully, rare.

Cruftable obfuscation



```
;-- section..text.maven_realloc:
(fcn) sym.maven_realloc 266
sym.maven_realloc (size_t size, int arg2, int arg3);
; arg size_t size @ rdi
; arg int arg2 @ rsi
; arg int arg3 @ rdx
; XREFS: CALL 0x0800080d  CALL 0x08000861  CALL 0x0800087b  CALL 0x08000889  CALL 0x08000895  CALL 0x080008b5
0x080007d0      nop dword [rax + rax]                                   ; [33] -r-x section size 266 named .text.maven_reall
0x080007d5      push r15
0x080007d7      push r14
0x080007d9      push r13
0x080007db      mov r13, rsi                                            ; arg2
0x080007de      push r12
0x080007e0      push rbp
0x080007e1      push rbx
0x080007e2      test rdi, rdi
0x080007e5      je 0x80008c8
```

```
; CODE XREF from 0x080007e5 (sym.maven_realloc)
0x080008c8      pop rbx
0x080008c9      mov rdi, rsi
0x080008cc      pop rbp
0x080008cd      pop r12
0x080008cf      pop r13
0x080008d1      pop r14
0x080008d3      pop r15
(reloc.maven_alloc_214)
0x080008d5      jmp entry0                                   ; RELOC 32 maven_allo
```

```
0x080007eb      mov rbp, rdi                                            ; arg1
0x080007ee      xor ebx, ebx
0x080007f0      xor r12d, r12d
0x080007f3      mov r14, 0                                              ; RELOC 32
```

```
; CODE XREF from 0x08000822 (sym.maven_realloc)
0x080007fa      mov rdi, qword [r14 + rbx*8]                            ; void *ptr
0x080007fe      xor edx, edx
0x08000800      mov rax, rbp
0x08000803      mov rsi, rbp
0x08000806      div qword [rdi + 8]
0x0800080a      sub rsi, rdx                                            ; size_t size
(reloc.mvn_addrinpool_14)
0x0800080d      call 0x8000812                                          ; RELOC 32 mvn_addrinpool ; void *realloc(void *ptr, size_t siz
; CALL XREF from 0x0800080d (sym.maven_realloc)
0x08000812      test eax, eax
0x08000814      je 0x800081a
```

```
0x08000816      mov r12, qword [r14 + rbx*8]
```

```
; CODE XREF from 0x08000814 (sym.maven_realloc)
0x0800081a      add rbx, 1
0x0800081e      cmp rbx, 6                                              ;
0x08000822      jne 0x80007fa
```

```
0x08000824      test r12, r12
0x08000827      je 0x800082d                                            ; RELOC 32
```

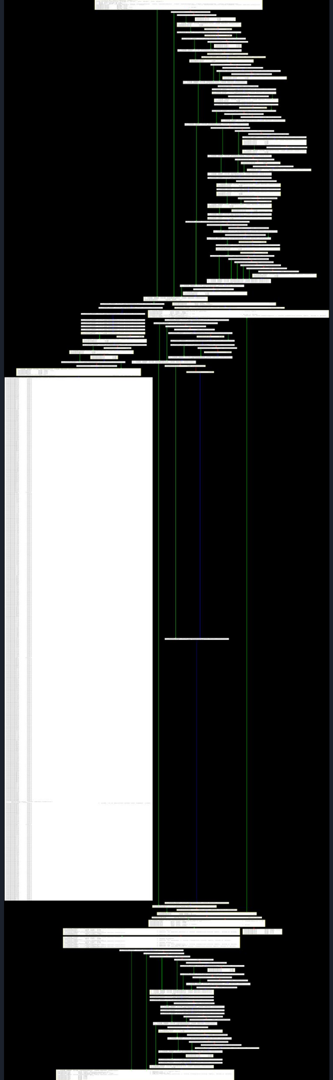
```
; CODE XREF from 0x08000827 (sym.maven_realloc)
0x0800082d      mov rcx, qword [r12 + 8]                                ; [0x8:8]=-1 ;
0x08000832      mov rax, rbp
0x08000835      xor edx, edx
0x08000837      mov r14, rbp
0x0800083a      div rcx
0x0800083d      lea rax, [r13 + rdx]
0x08000842      mov r15, rdx                                            ; arg3
0x08000845      cmp rcx, rax
0x08000848      jbe 0x8000858
```

```
; CODE XREF from 0x08000848 (sym.maven_realloc)
0x08000858      sub rcx, rdx
0x0800085b      mov rdi, r13                                            ; void *ptr
0x0800085e      mov rbx, rcx
(reloc.maven_alloc_98)
0x08000861      call 0x8000866                                          ; RELOC 32 maven_alloc ; void *realloc(void *ptr, size_t siz
; CALL XREF from 0x08000861 (sym.maven_realloc)
0x08000866      mov r14, rax
0x08000869      test rax, rax
0x0800086c      je 0x8000872                                            ; RELOC 32
```

```
0x0800084a      pop rbx
0x0800084b      mov rax, r14
0x0800084e      pop rbp
0x0800084f      pop r12
0x08000851      pop r13
0x08000853      pop r14
0x08000855      pop r15
0x08000857      ret
```

```
; CODE XREF from 0x0800086c (sym.maven_realloc)
0x08000872      mov rdx, rbx
0x08000875      mov rsi, rbp                                            ; size_t size
0x08000878      mov rdi, rax                                            ; void *ptr
(reloc.memcpy)
0x0800087b      call 0x8000880                                          ; RELOC 32 memcpy ; void *realloc(void *ptr, size_t size)
; CALL XREF from 0x0800087b (sym.maven_realloc)
0x08000880      mov rsi, rbp
0x08000883      mov rdi, r12                                            ; void *ptr
0x08000886      sub rsi, r15                                            ; size_t size
(reloc.mvn_memfree_138)
0x08000889      call 0x800088e                                          ; RELOC 32 mvn_memfree ; void *realloc(void *ptr, size_t size
; CALL XREF from 0x08000889 (sym.maven_realloc)
0x0800088e      mov rdi, 0                                              ; RELOC 32  ; void *ptr
(reloc.mutex_lock)
0x08000895      call 0x800089a                                          ; RELOC 32 mutex_lock ; void *realloc(void *ptr, size_t size)
; CALL XREF from 0x08000895 (sym.maven_realloc)
0x0800089a      mov rdi, 0                                              ; RELOC 32  ; void *ptr
0x080008a1      mov rax, qword [r12 + 8]                                ; [0x8:8]=-1 ; 8
0x080008a6      add qword [0x080008ae], 1                               ; RELOC 32
0x080008ae      add qword [0x080008b5], rax                             ; '#'; RELOC 32
(reloc.mutex_unlock)
0x080008b5      call 0x80008ba                                          ; RELOC 32 mutex_unlock ; void *realloc(void *ptr, size_t siz
; CALL XREF from 0x080008b5 (sym.maven_realloc)
0x080008ba      mov rax, r14
0x080008bd      pop rbx
0x080008be      pop rbp
0x080008bf      pop r12
0x080008c1      pop r13
0x080008c3      pop r14
0x080008c5      pop r15
0x080008c7      ret
```

Cruftable obfuscation

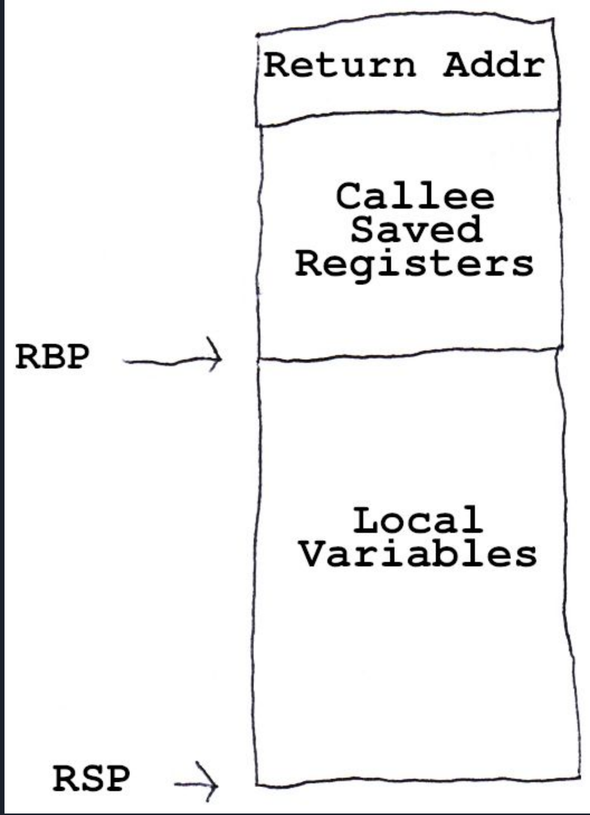


Stack Stuff

- Stack shims
  - Insert a “shim” into each stack frame
    - Changes stack overflow and leak offsets
    - Protects against small overflows
- Stack reordering
  - Reorder Pushes and Pops of saved registers.
    - Changes ROP gadgets.
    - Can change leak offsets.

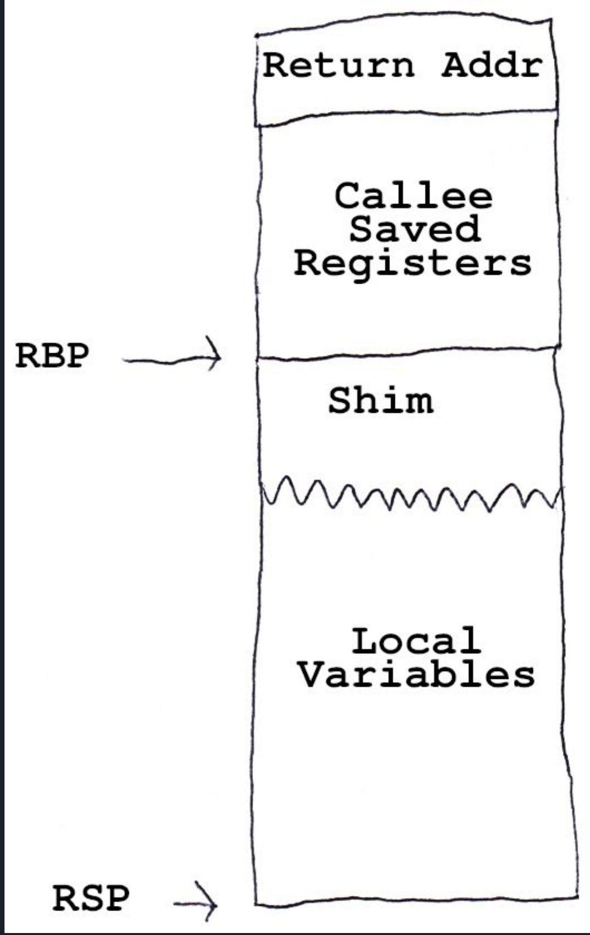
Stack Shims

- Adjust allocated stack frame size.
  - Add a little “buffer” area on the end.
  - Makes stack-based OOB reads or writes unreliable
  - May even protect against small OOB writes.



Stack Shims

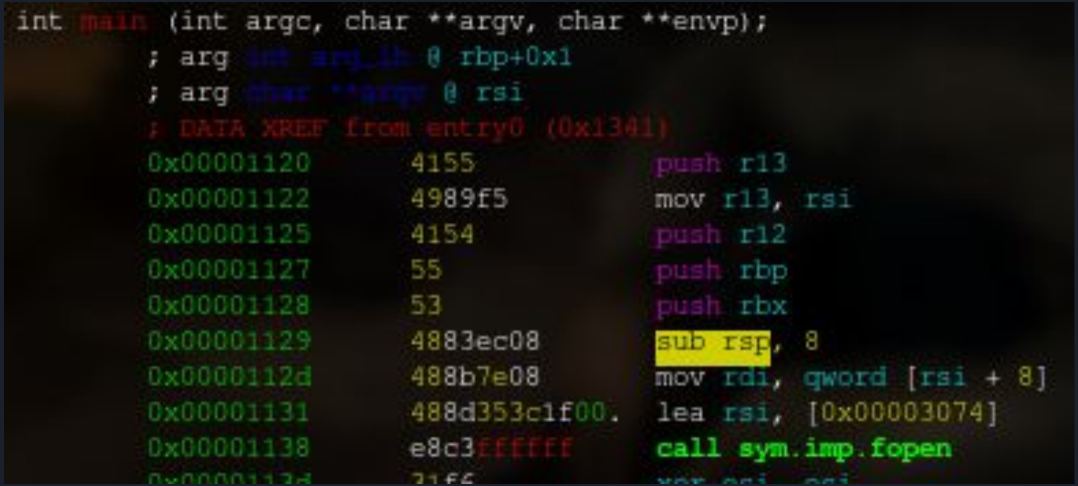
- Adjust allocated stack frame size.
  - Add a little “buffer” area in frame.
  - Makes stack-based OOB reads or writes unreliable
  - May even protect against small OOB writes.



Stack Shim size alignment.

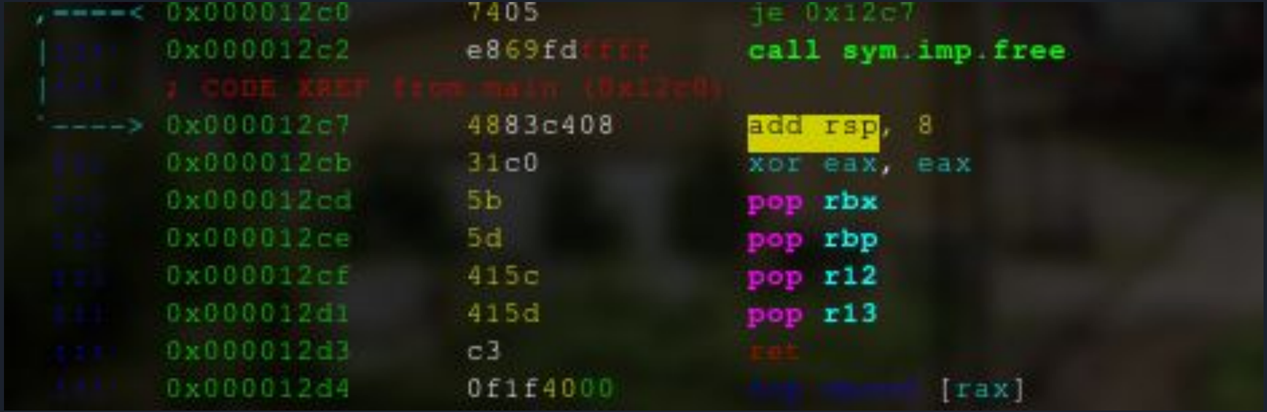
- May have to be aligned to 16 or 8 bytes.
    - GCC defaults to align to 16.
- Some SSE/SIMD instructions will segfault if not aligned.
- Performance impact from unaligned memory accesses.
- **Not always required.**

Stack frame allocation.



```
int main (int argc, char **argv, char **envp);
        ; arg int arg_1h @ rbp+0x1
        ; arg char **argv @ rsi
        ; DATA XREF from entry0 (0x1341)
        0x00001120      4155            push r13
        0x00001122      4989f5          mov r13, rsi
        0x00001125      4154            push r12
        0x00001127      55              push rbp
        0x00001128      53              push rbx
        0x00001129      4883ec08        sub rsp, 8
        0x0000112d      488b7e08        mov rdi, qword [rsi + 8]
        0x00001131      488d353c1f00.   lea rsi, [0x00003074]
        0x00001138      e8c3ffffff      call sym.imp.fopen
        0x0000113d      31f6            xor esi, esi
```

Stack frame deallocation



```
,=====< 0x000012c0      7405            je 0x12c7
|:::    0x000012c2      e869fdffff      call sym.imp.free
|:::    ; CODE XREF from main (0x12c0)
`----> 0x000012c7      4883c408        add rsp, 8
 :::    0x000012cb      31c0            xor eax, eax
 :::    0x000012cd      5b              pop rbx
 :::    0x000012ce      5d              pop rbp
 :::    0x000012cf      415c            pop r12
 :::    0x000012d1      415d            pop r13
 :::    0x000012d3      c3              ret
 :::    0x000012d4      0f1f4000        nop dword [rax]
```

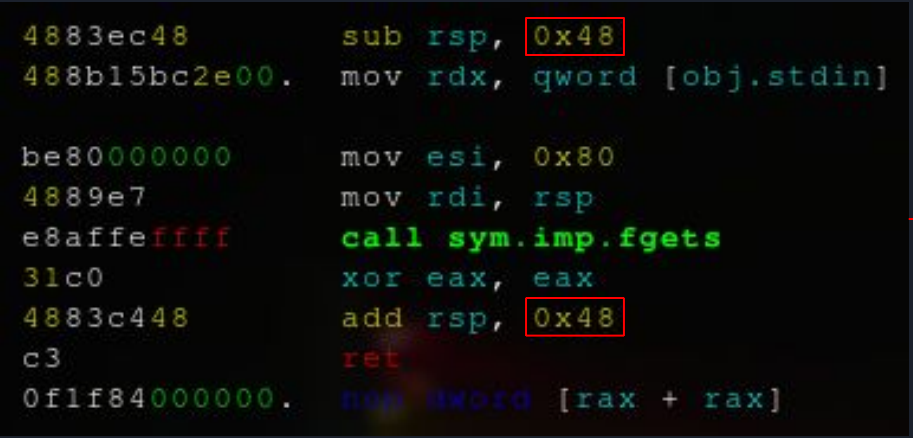
Multiple Deallocations

- Each allocation may have multiple deallocations
    - For allocation in function (symbol), look for all deallocations that match its size.
        - Potential for inconsistencies here, but we’ve never encountered one in our testing.

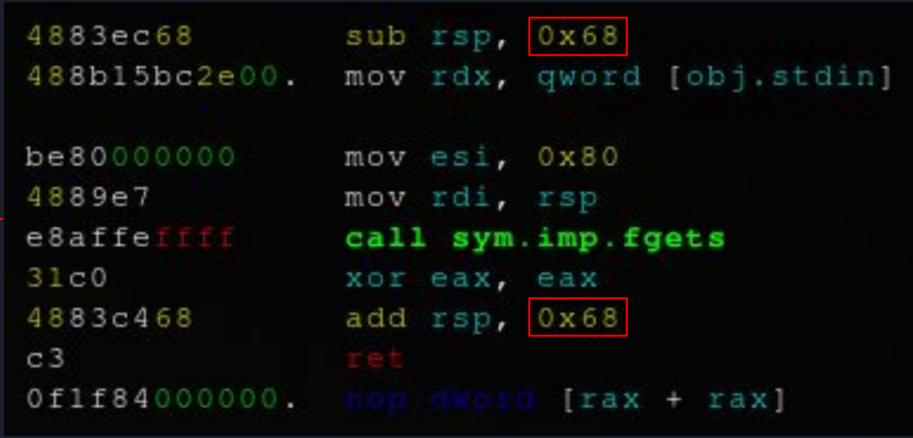
Some assumptions

- Only one stack frame per function.
  - Would be difficult to track multiple frames, especially with odd control flow.
  - GCC is nice and does one frame per function (symbol).
- Allocations and deallocations are symmetric
  - No half-allocate or half-deallocate.
  - GCC is also nice here.

Results



```
4883ec48          sub rsp, 0x48
488b15bc2e00.     mov rdx, qword [obj.stdin]

be80000000        mov esi, 0x80
4889e7            mov rdi, rsp
e8affeffff        call sym.imp.fgets
31c0              xor eax, eax
4883c448          add rsp, 0x48
c3                ret
0f1f84000000.     nop dword [rax + rax]
```



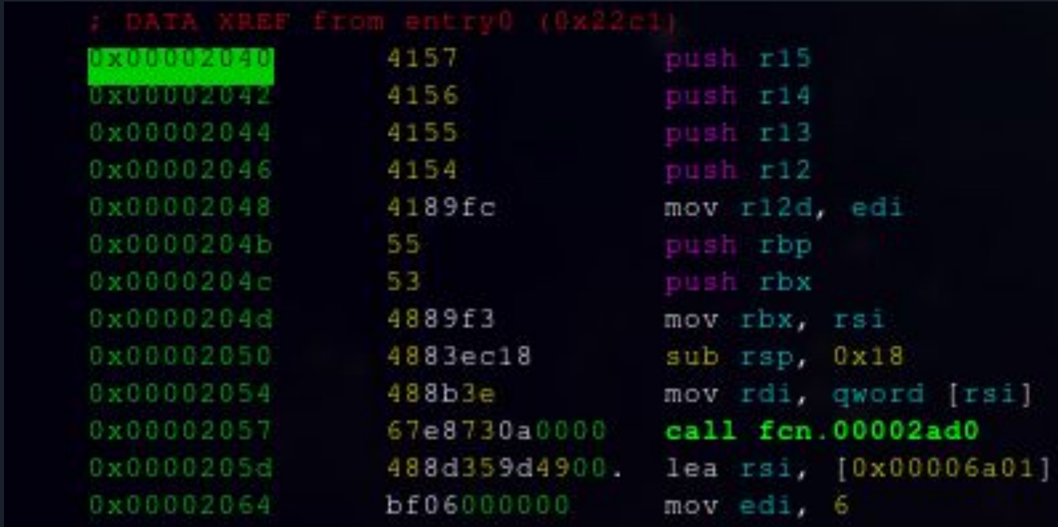
```
4883ec68          sub rsp, 0x68
488b15bc2e00.     mov rdx, qword [obj.stdin]

be80000000        mov esi, 0x80
4889e7            mov rdi, rsp
e8affeffff        call sym.imp.fgets
31c0              xor eax, eax
4883c468          add rsp, 0x68
c3                ret
0f1f84000000.     nop dword [rax + rax]
```

Stack Reordering

Reorder the pushes and pops of saved registers.

Two places to reorder

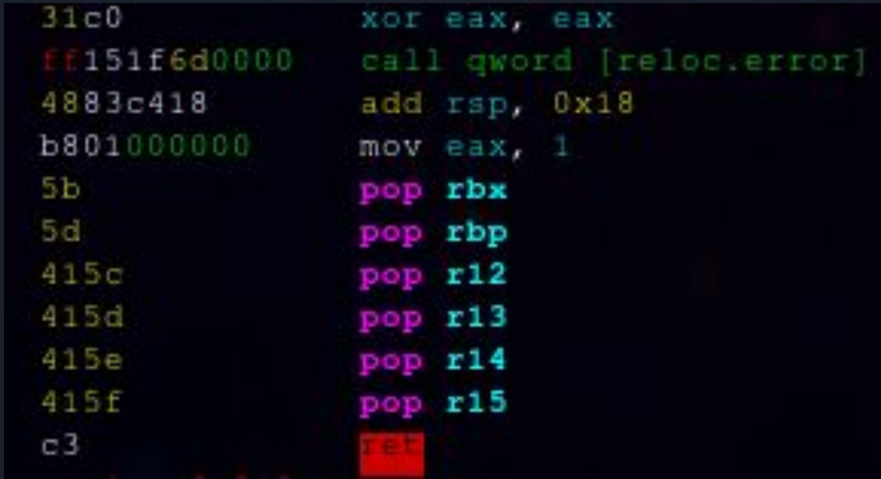
- Prologue
    - Only one, the start of the function. (Symbol)
    - Lots of Pushes



```
; DATA XREF from entry0 (0x22c1)
0x00002040      4157            push r15
0x00002042      4156            push r14
0x00002044      4155            push r13
0x00002046      4154            push r12
0x00002048      4189fc          mov r12d, edi
0x0000204b      55              push rbp
0x0000204c      53              push rbx
0x0000204d      4889f3          mov rbx, rsi
0x00002050      4883ec18        sub rsp, 0x18
0x00002054      488b3e          mov rdi, qword [rsi]
0x00002057      67e8730a0000    call fcn.00002ad0
0x0000205d      488d359d4900.   lea rsi, [0x00006a01]
0x00002064      bf06000000      mov edi, 6
```

Two places to reorder

- Epilogue(s)
    - Can be multiple
    - Right before a “ret”
    - Lots of Pops
    - May not be directly at the end of a function.

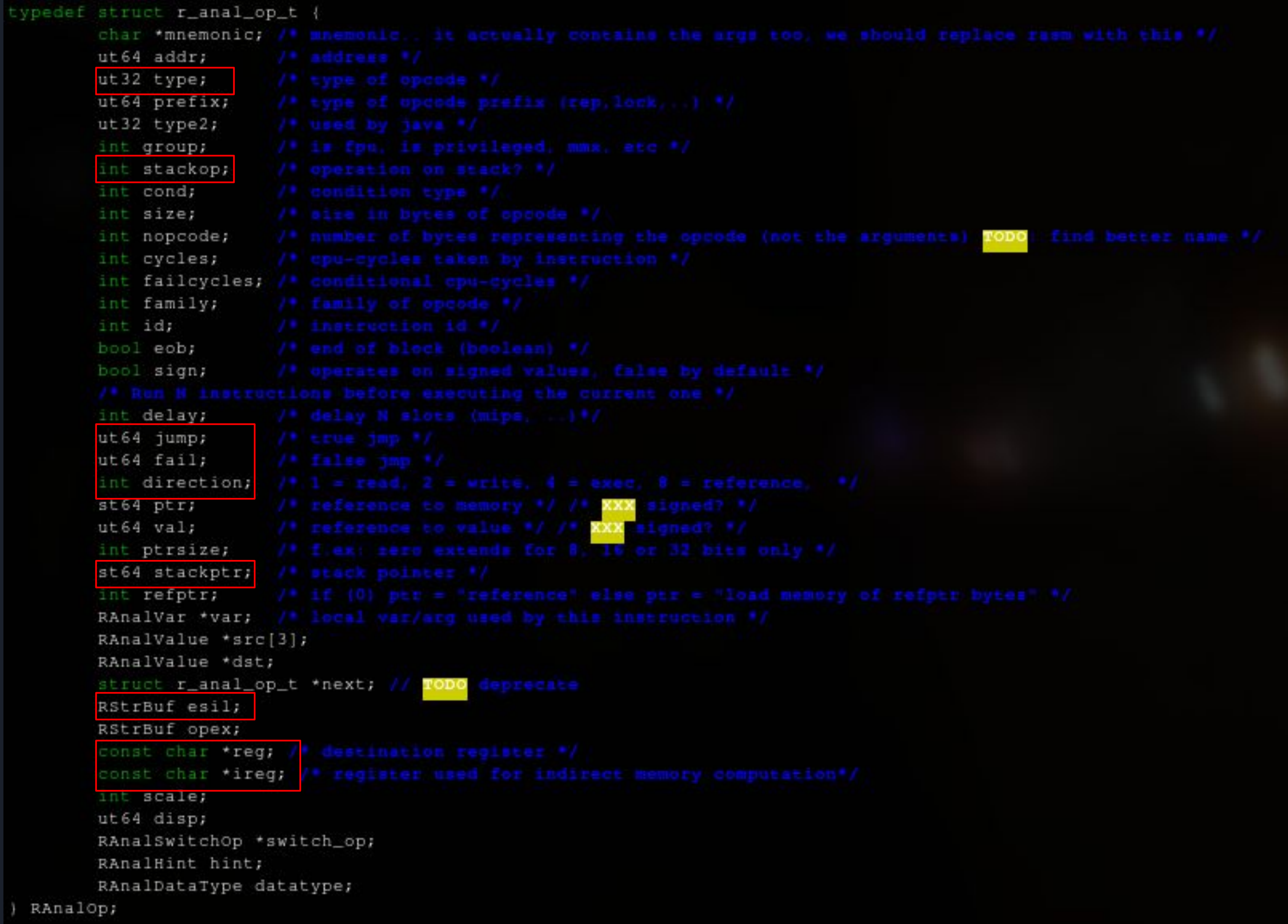


```
31c0              xor eax, eax
ff151f6d0000      call qword [reloc.error]
4883c418          add rsp, 0x18
b801000000        mov eax, 1
5b                pop rbx
5d                pop rbp
415c              pop r12
415d              pop r13
415e              pop r14
415f              pop r15
c3                ret
```

Look for reorderable instructions

- Requirements:
    - **No modifications to RIP** (Jumps, calls, etc)
    - No jumps pointing into the area (Look at badzones).
    - No relocs in the area (may be able to be changed in the future)
    - **No stack operations that aren’t push/pop.**
        - **strstr(op->esil, ’rsp’)**
    - **No memory Writes** (might be not required)
    - **Can not use a push if in a epilogue, can not use a pop if in an Prologue.**
    - Register pushes must match Register Pops.
        - Truncation step.

RAnalOp Is Super Useful. (Again)

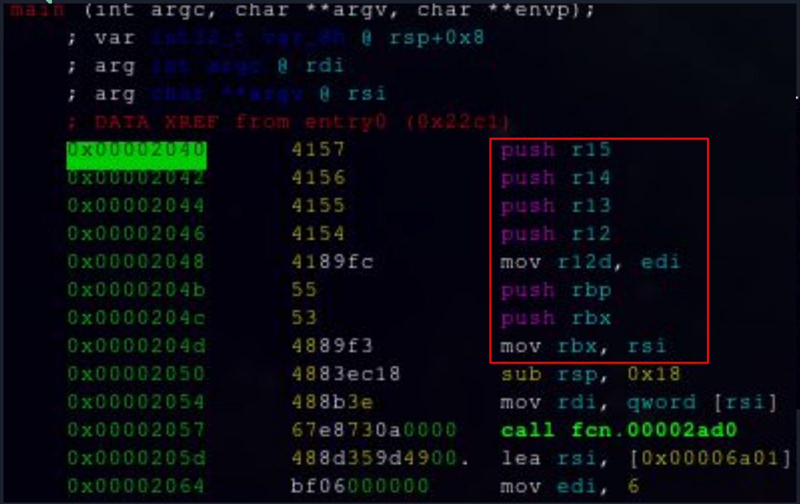


```
typedef struct r_anal_op_t {
        char *mnemonic; /* mnemonic.. it actually contains the args too, we should replace rasm with this */
        ut64 addr;      /* address */
        ut32 type;      /* type of opcode */
        ut64 prefix;    /* type of opcode prefix (rep,lock,..) */
        ut32 type2;     /* used by java */
        int group;      /* is fpu, is privileged, mmx, etc */
        int stackop;    /* operation on stack? */
        int cond;       /* condition type */
        int size;       /* size in bytes of opcode */
        int nopcode;    /* number of bytes representing the opcode (not the arguments) TODO: find better name */
        int cycles;     /* cpu-cycles taken by instruction */
        int failcycles; /* conditional cpu-cycles */
        int family;     /* family of opcode */
        int id;         /* instruction id */
        bool eob;       /* end of block (boolean) */
        bool sign;      /* operates on signed values, false by default */
        /* Run N instructions before executing the current one */
        int delay;      /* delay N slots (mips, ..)*/
        ut64 jump;      /* true jmp */
        ut64 fail;      /* false jmp */
        int direction;  /* 1 = read, 2 = write, 4 = exec, 8 = reference,  */
        st64 ptr;       /* reference to memory */ /* XXX signed? */
        ut64 val;       /* reference to value */ /* XXX signed? */
        int ptrsize;    /* f.ex: zero extends for 8, 16 or 32 bits only */
        st64 stackptr;  /* stack pointer */
        int refptr;     /* if (0) ptr = "reference" else ptr = "load memory of refptr bytes" */
        RAnalVar *var;  /* local var/arg used by this instruction */
        RAnalValue *src[3];
        RAnalValue *dst;
        struct r_anal_op_t *next; // TODO deprecate
        RStrBuf esil;
        RStrBuf opex;
        const char *reg; /* destination register */
        const char *ireg; /* register used for indirect memory computation*/
        int scale;
        ut64 disp;
        RAnalSwitchOp *switch_op;
        RAnalHint hint;
        RAnalDataType datatype;
} RAnalOp;
```

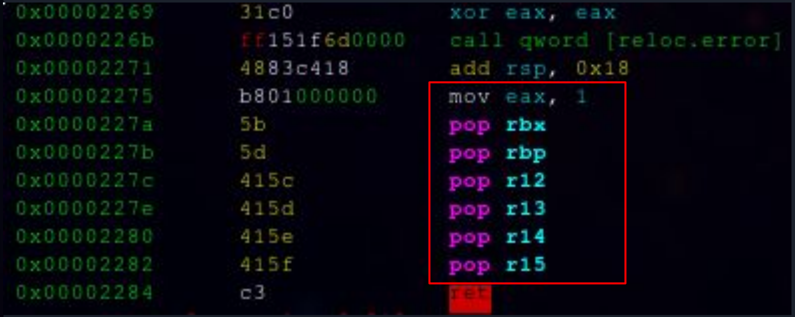
Truncation step

- If a Prologue/epilogue has an extra push or pop that isn’t matched in ALL of the others, we must truncate it out.
    - Rare, but can happen.

What do we have left?



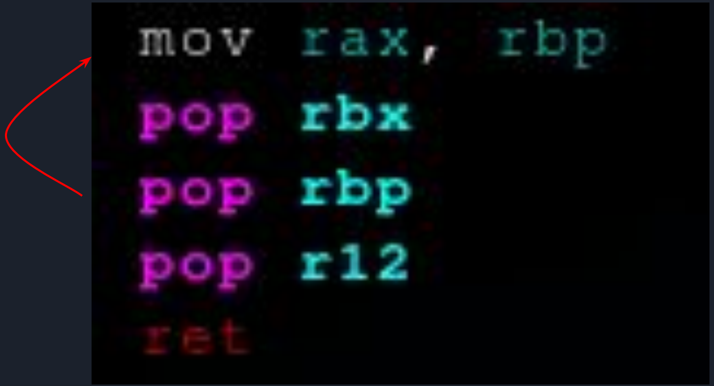
```
main (int argc, char **argv, char **envp);
     ; var int32_t var_8h @ rsp+0x8
     ; arg int argc @ rdi
     ; arg char **argv @ rsi
     ; DATA XREF from entry0 (0x22c1)
     0x00002040          4157              push r15
     0x00002042          4156              push r14
     0x00002044          4155              push r13
     0x00002046          4154              push r12
     0x00002048          4189fc            mov r12d, edi
     0x0000204b          55                push rbp
     0x0000204c          53                push rbx
     0x0000204d          4889f3            mov rbx, rsi
     0x00002050          4883ec18          sub rsp, 0x18
     0x00002054          488b3e            mov rdi, qword [rsi]
     0x00002057          67e8730a0000      call fcn.00002ad0
     0x0000205d          488d359d4900.     lea rsi, [0x00006a01]
     0x00002064          bf06000000        mov edi, 6
```



```
0x00002269          31c0              xor eax, eax
0x0000226b          ff151f6d0000      call qword [reloc.error]
0x00002271          4883c418          add rsp, 0x18
0x00002275          b801000000        mov eax, 1
0x0000227a          5b                pop rbx
0x0000227b          5d                pop rbp
0x0000227c          415c              pop r12
0x0000227e          415d              pop r13
0x00002280          415e              pop r14
0x00002282          415f              pop r15
0x00002284          c3                ret
```

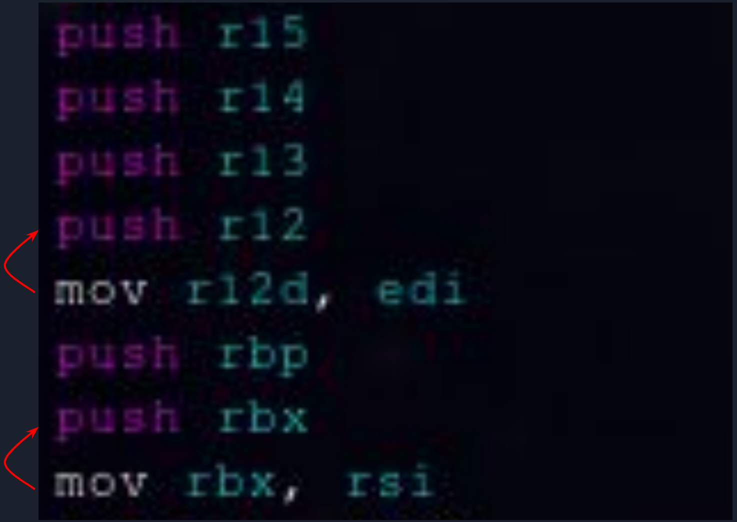
Build dependency list

- Each instruction has a list of dependency instructions that must come before.
  - For every instruction, see what instructions comes before it that use any of the same registers.
  - Instructions may have many dependencies, but cycles are not possible.
- When shuffling, instruction must have all of its dependencies fulfilled before being chosen.
  - (Fisher-yates)
- Seperate lists for every epilogue/prologue.



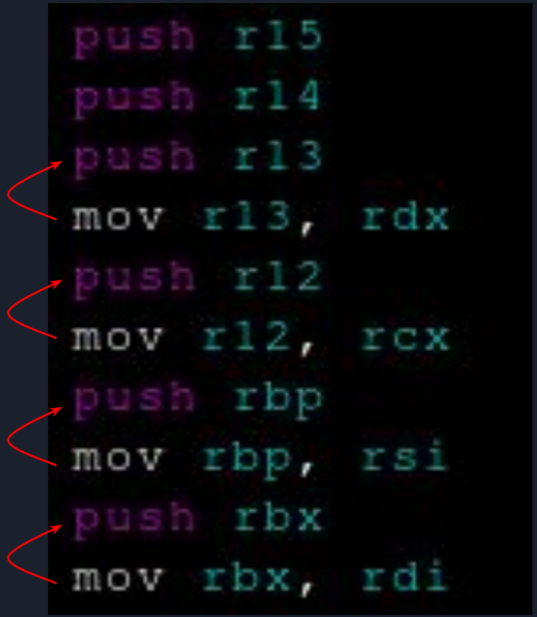
```
mov rax, rbp
pop rbx
pop rbp
pop r12
ret
```

Build dependency list

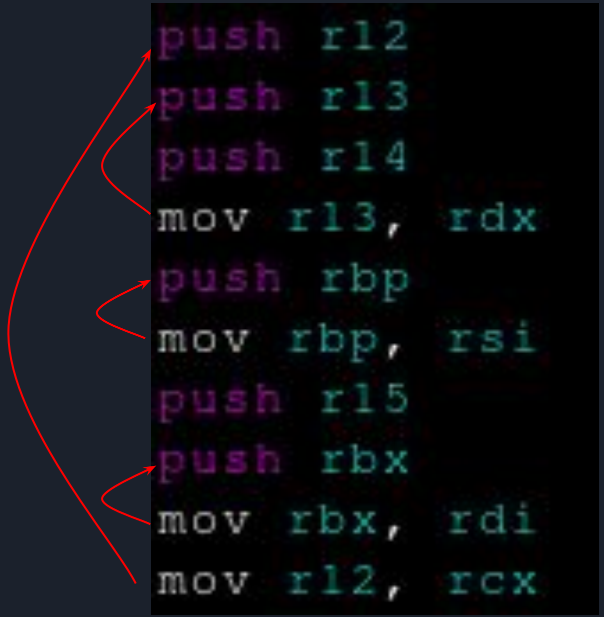


```
push r15
push r14
push r13
push r12
mov r12d, edi
push rbp
push rbx
mov rbx, rsi
```

Dependencies kept after shuffling



```
push r15
push r14
push r13
mov r13, rdx
push r12
mov r12, rcx
push rbp
mov rbp, rsi
push rbx
mov rbx, rdi
```

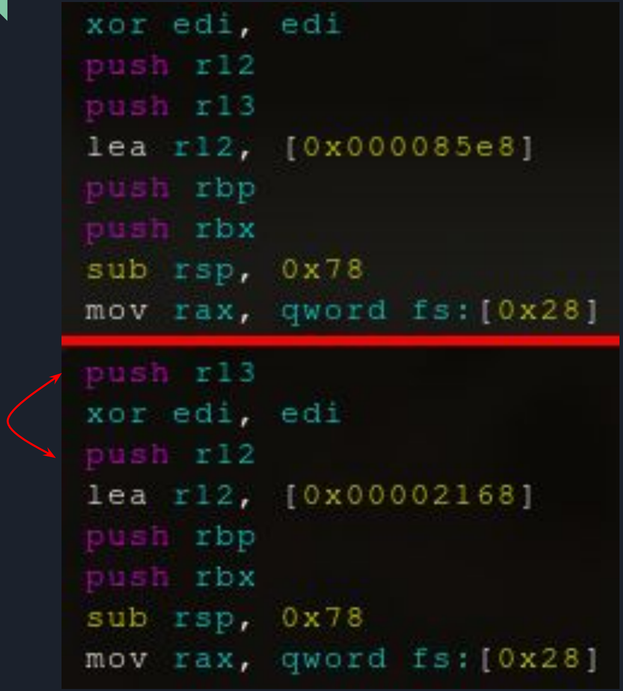


```
push r12
push r13
push r14
mov r13, rdx
push rbp
mov rbp, rsi
push r15
push rbx
mov rbx, rdi
mov r12, rcx
```

Prologue/Epilogue Symmetry

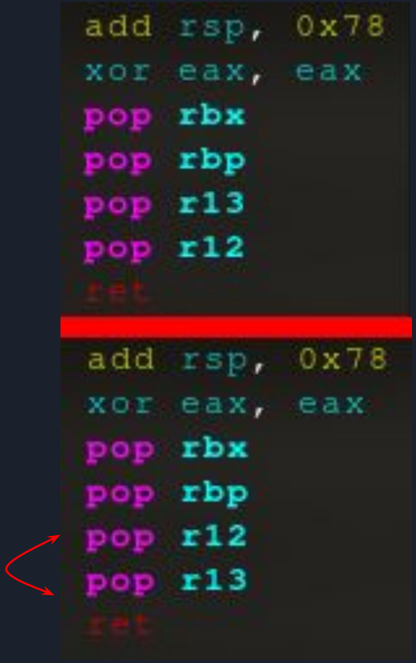
- Whatever order we shuffle the Pushes in the prologue, we must change the Pops in Epilogue to match.
  - Reverse order.
  - Only pushes/pops need to follow order. Misc instructions are fine to reorder.
  - Still have to follow instruction dependencies too.

Prologue/Epilogue Symmetry



```
xor edi, edi
push r12
push r13
lea r12, [0x000085e8]
push rbp
push rbx
sub rsp, 0x78
mov rax, qword fs:[0x28]
```

```
push r13
xor edi, edi
push r12
lea r12, [0x00002168]
push rbp
push rbx
sub rsp, 0x78
mov rax, qword fs:[0x28]
```



```
add rsp, 0x78
xor eax, eax
pop rbx
pop rbp
pop r13
pop r12
ret
```

```
add rsp, 0x78
xor eax, eax
pop rbx
pop rbp
pop r12
pop r13
ret
```

Demo

Future things

- Use R2 emulation functionality for fancier analysis/verification.
  - Would allow for more complex modifications and higher reliability.
- Port to ARM, Mips, Risc-V, etc.
- Full binary support.
  - Might be difficult due to linker resolving some symbols.
- Port to PE, Mach-O, etc.
- More advanced cruftable data.
  - Chunks of code that actually look like code.
  - More complex structure.
  - Harder to automatically de-obfuscate.
- Shuffle at binary/object load time
  - Kernel module?
  - It’s fast enough for this to be feasible.
- Standalone “Patch System” as a plugin for R2.

Special thanks,

Martin Osterloh for section shuffling.

Jesse Earisman for most of the ELF code.

R2 devs for answering all of my noob questions and fixing bugs.

Siege Technologies for sending me to R2Con.

Questions?

Unused/unfinished slides

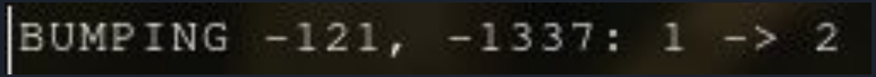
Jumptable hack

GCC is “smart” and does jumptables in a weird way.

Todo

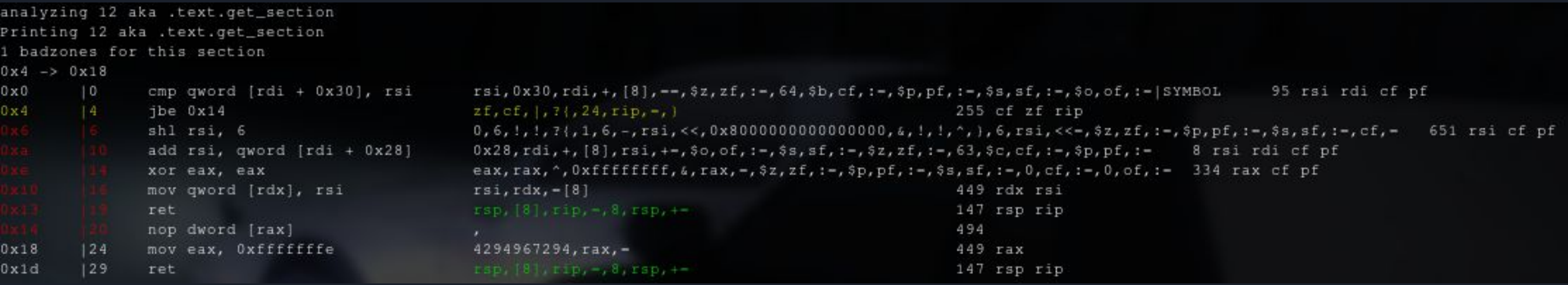
Instruction Bumping

- Convert short jumps to near jumps. (if needed)
- Processed recursively by patch system.
  - Bumping one instruction may cause another instruction to be bumped.



```
BUMPING -121, -1337: 1 -> 2
```

Analysis output



```
analyzing 12 aka .text.get_section
Printing 12 aka .text.get_section
1 badzones for this section
0x4 -> 0x18
0x0     |0      cmp qword [rdi + 0x30], rsi      rsi,0x30,rdi,+,[8],==,$z,zf,:=,64,$b,cf,:=,$p,pf,:=,$s,sf,:=,$o,of,:=|SYMBOL     95 rsi rdi cf pf
0x4     |4      jbe 0x14                         zf,cf,|,?{,24,rip,=,}                                    255 cf zf rip
0x6     |6      shl rsi, 6                       0,6,!,!,?{,1,6,-,rsi,<<,0x8000000000000000,&,!,!,^,},6,rsi,<<=,$z,zf,:=,$p,pf,:=,$s,sf,:=,cf,=   651 rsi cf pf
0xa     |10     add rsi, qword [rdi + 0x28]      0x28,rdi,+,[8],rsi,+=,$o,of,:=,$s,sf,:=,$z,zf,:=,63,$c,cf,:=,$p,pf,:=    8 rsi rdi cf pf
0xe     |14     xor eax, eax                     eax,rax,^,0xffffffff,&,rax,=,$z,zf,:=,$p,pf,:=,$s,sf,:=,0,cf,:=,0,of,:=  334 rax cf pf
0x10    |16     mov qword [rdx], rsi             rsi,rdx,=[8]                                             449 rdx rsi
0x13    |19     ret                              rsp,[8],rip,=,8,rsp,+=                                   147 rsp rip
0x14    |20     nop dword [rax]                  ,                                                        494
0x18    |24     mov eax, 0xfffffffe              4294967294,rax,=                                         449 rax
0x1d    |29     ret                              rsp,[8],rip,=,8,rsp,+=                                   147 rsp rip
```

Two types of relative addressing

- RSP and RBP relative.
    - RSP+offset
        - for local variables.
    - RBP+offset
        - for stack arguments.
- (Ideally)

GCC sometimes does gross stuff

And this changes from version to version.

- RSP + to grab stack arguments.
    - “Reaches over rbp”
- RBP- relative addressing for local vars.

Addressing Fixes

- Convert the “bad” modes to the appropriate one. - Will not work in all cases.
  - If RBP is used as general purpose, can’t convert to RBP-relative.
  - May need to do some tracking of pushes/pops between frame allocation and usage.
- Add the stack shim offset into the bad modes. - More likely to work.
  - Needs to be done to every instruction that “Reaches over”

We currently use the first method. Not 100% reliable.

GCC Sibling calls

- GCC for optimization will make “sibling calls”
  - Tail-call optimization
  - Instead of a call/ret, just use a JMP.
- Or sometimes will split a function in two.
- Force to show up as a “epilogues”.
  - There is no ret.
  - May not fully pop stuff from stack. Symmetry/truncation will handle it.