ELEC 334 - Homework #3

Berat KIZILARMUT - 171024086

A. Problem 1 – Function calls

Parameters need to be loaded to registers. Registers used depends on the parameters of the said C function. Basic principle is starting from a single parameter it descends from r3 to r0 with each added parameter. When we try to use 5 parameters we need to start storing or pushing the parameters

a. Single Parameter, starts from r3.

```
LDR r3, par1
BL 0 <func>
```

```
void func(int par1)
```

b. Two Parameters, adds r2

```
LDR r2, par1
LDR r3, par2
BL <func>
```

```
void func(int par1, int par2)
```

c. Three Parameters, adds r1

```
LDR r1, par1
LDR r2, par2
LDR r3, par3
BL <func>
```

```
void func(int par1, int par2, int par3)
```

d. Four Parameters, add r0

```
LDR r0, par1
LDR r1, par2
LDR r2, par3
LDR r3, par4
BL <func>
```

```
void func(int par1, int par2, int par3, int par4)
```

e. Five Parameters, pushing the fifth parameter to stack

```
LDR r3, par5
PUSH {r3}
LDR r0, par1
LDR r1, par2
LDR r2, par3
LDR r3, par4
BL <func>
```

```
void func(int par1, int par2, int par3, int par4, int par5)
```

f. Six Parameters, switching the parameter5 to r2 and following the same concept as prior

```
LDR r2, par5
LDR r3, par6
PUSH {r3}
PUSH {r2}
LDR r0, par1
LDR r1, par2
LDR r2, par3
LDR r3, par4
BL <func>
```

```
void func(int par1, int par2, int par3, int par4, int par5, int par6)
```

B. Problem 2 – Return values

Returned value from the function should be stored in EAX, general purpose register, and moved to the desired register to use.

```
...
func:
...
MOV eax, x
...

BL func
MOV res, eax

...
...
```

```
int func(int a) {
    ...
    return x;
}

main() {
    ...
    int s;
    s = func(a)
    ...
}
```

C. Problem 3 – Reverse me if you can

Provided “2020-hw3.elf” file has been disassembled by “GNU Arm Embedded Toolchain” using “arm-none-eabi-objdump -D” command, result given below. Added comments to what I’ve understood from the disassembled file.

```
Microsoft Windows [Version 10.0.18363.1198]
(c) 2019 Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\kuros>arm-none-eabi-objdump -D Desktop\2020-hw3.elf

Desktop\2020-hw3.elf:     file format elf32-littlearm


Disassembly of section .text:

08000000 <v>:
 8000000:       10002000        andne   r2, r0, r0
 8000004:       08000021        stmdaeq r0, {r0, r5} ;Store multiple increments, Decrement Address
 8000008:       0800002b        stmdaeq r0, {r0, r1, r3, r5} ; Same op, including r1 and r3 as well
 800000c:       0800002b        stmdaeq r0, {r0, r1, r3, r5} ; Same op
 8000010:       10000000        andne   r0, r0, r0 ;Does nothing
 8000014:       10000000        andne   r0, r0, r0
 8000018:       10000000        andne   r0, r0, r0
 800001c:       10000000        andne   r0, r0, r0

08000020 <r>:
 8000020:       481b            ldr     r0, [pc, #108]  ; (8000090 <lizard+0x10>) Goes to 8000020 + 108 = 08000080
 8000022:       4685            mov     sp, r0 ;moves that address to stack pointer
 8000024:       f000 f802       bl      800002c <main> ;branch links to main
 8000028:       e7fe            b.n     8000028 <r+0x8> ;indefinite loop

0800002a <d>:
 800002a:       e7fe            b.n     800002a <d> ;indefinite loop

0800002c <main>:
 800002c:       4919            ldr     r1, [pc, #100]  ; (8000094 <lizard+0x14>)  Goes to 800002c + 100 = 8000094
 800002e:       4a1a            ldr     r2, [pc, #104]  ; (8000098 <lizard+0x18>)   Goes to 800002e + 104 = 8000098
 8000030:       2300            movs    r3, #0 ; Move zero to r3

08000032 <rock>:
 8000032:       f000 f807       bl      8000044 <paper> ;Branch link to paper
 8000036:       6010            str     r0, [r2, #0] ;Writes value of the address that r2 points to r0
 8000038:       3104            adds    r1, #4 ;Adds 4 to r1
 800003a:       3204            adds    r2, #4 ;Adds 4 to r2
 800003c:       3301            adds    r3, #1 ;Adds 4 to r3
 800003e:       2b04            cmp     r3, #4 ;Compare r3 to 4
 8000040:       d1f7            bne.n   8000032 <rock> ;Branch to rock if Not Equal
 8000042:       e017            b.n     8000074 <eof>   ;Branch to end of function

08000044 <paper>:
 8000044:       b40e            push    {r1, r2, r3} ;Push r1 r2 and r3 to stack
 8000046:       4e15            ldr     r6, [pc, #84]   ; (800009c <lizard+0x1c>) Goes to 8000044 + 84 = 800009c
 8000048:       00f7            lsls    r7, r6, #3 ;Logic Shift Left r6 by 3 and write to r7, 400004e0
 800004a:       6809            ldr     r1, [r1, #0] ;Writes value of the address that r1 points to r1
 800004c:       4c14            ldr     r4, [pc, #80]   ; (80000a0 <lizard+0x20>) Goes to 800004c + 80 = 80000a0

0800004e <scissors>:
 800004e:       4a15            ldr     r2, [pc, #84]   ; (80000a4 <lizard+0x24>) Gets the program counter + 84 offsett address to r2
 8000050:       6815            ldr     r5, [r2, #0] ;Writes value of the address that r2 points to r5
 8000052:       0108            lsls    r0, r1, #4 ;Logic Shift Left r1 by 4 and write to r0
 8000054:       1940            adds    r0, r0, r5 ;Adds r0 r5 and writes back to r0
 8000056:       b401            push    {r0} ;Pushes r0 to stack
 8000058:       6855            ldr     r5, [r2, #4] ;Writes value of the address that r2 points plus 4 to r4
 800005a:       0948            lsrs    r0, r1, #5 ;Logic Shift Right r1 by 5 and write to r0
 800005c:       1940            adds    r0, r0, r5 ;Add r5 to r0 and write to r0
 800005e:       19ca            adds    r2, r1, r7 ;Add r7 to r1 and write to r2
 8000060:       4050            eors    r0, r2 ;XOR r2 and write to r0
 8000062:       bc04            pop     {r2} ;POP from stack and write to r2
 8000064:       4050            eors    r0, r2 ;XOR r2 and write to r0
 8000066:       1a09            subs    r1, r1, r0 ;Subtract r0 from r1 and write to r1
 8000068:       1bbf            subs    r7, r7, r6  ;Subtract r6 from r7 and write to r7
 800006a:       0864            lsrs    r4, r4, #1 ;Logic Shift Right r4 by 4 and write back to r4
 800006c:       d1ef            bne.n   800004e <scissors> ;Depending on the flag conditions, loop back to scissors
 800006e:       0008            movs    r0, r1 ;Move r1 to r0
 8000070:       bc0e            pop     {r1, r2, r3} ;POP r1 r2 and r3 to stack
 8000072:       4770            bx      lr ;branch and exchange operation, causes branching to an instruction set

08000074 <eof>: ;   End of function, loops infinitely
 8000074:       e7fe            b.n     8000074 <eof>
 8000076:       46c0            nop                     ; (mov r8, r8)

08000078 <spock>:
 8000078:       138a5b9c        orrne   r5, sl, #156, 22        ; 0x27000
 800007c:       83b19de5                        ; <UNDEFINED> instruction: 0x83b19de5

08000080 <lizard>:
 8000080:       a2390c55        eorsge  r0, r9, #21760  ; 0x5500
 8000084:       113f39fc        teqne   pc, ip  ; <illegal shifter operand>     ; <UNPREDICTABLE>
```

```
 8000088:       6140f4fd        strdvs  pc, [r0, #-77]  ; 0xffffffb3
 800008c:       d3926c34        orrsle  r6, r2, #52, 24 ; 0x3400
 8000090:       10002000        andne   r2, r0, r0
 8000094:       08000080        stmdaeq r0, {r7}
 8000098:       10000200        andne   r0, r0, r0, lsl #4
 800009c:       14159265        ldrne   r9, [r5], #-613 ; 0xfffffd9b
 80000a0:       00000080        andeq   r0, r0, r0, lsl #1
 80000a4:       08000078        stmdaeq r0, {r3, r4, r5, r6}

Disassembly of section .ARM.attributes:

00000000 <.ARM.attributes>:
   0:   00002141        andeq   r2, r0, r1, asr #2
   4:   61656100        cmnvs   r5, r0, lsl #2
   8:   01006962        tsteq   r0, r2, ror #18
   c:   00000017        andeq   r0, r0, r7, lsl r0
  10:   726f4305        rsbvc   r4, pc, #335544320      ; 0x14000000
  14:   2d786574        cfldr64cs       mvdx6, [r8, #-464]!     ; 0xfffffe30
  18:   002b304d        eoreq   r3, fp, sp, asr #32
  1c:   4d070c06        stcmi   12, cr0, [r7, #-24]     ; 0xffffffe8
  20:   Address 0x00000020 is out of bounds.
```

D. Problem 4 – Reading "*Reading Faults, Injection Methods, and Fault Attacks"*

Fault attacks are a type of embedded system hacking method that attempts to cause faulty results from the system and tries to extract information about the said system in the process. There are different ways to induce this attack on a system, the way they initiated, the type of fault they achieve to have. Some of these attack types are described briefly as;

Glitch attacks attempt to cause glitches in the system by trying to interfere with its internal clock or power dynamics. These attacks target the system at general and cannot be targeted at specifics aspects of a system.

Temperature attacks attempt to cause failures by extreme high or low temperatures to cause faults. By these extreme temperatures memory operation faults can occur.

Light attacks, unlike the prior two attack type, are very precise attacks. These attacks use some kind of a light, flash, laser etc. to attack photo sensitive parts of a system. These light's will affect the internal currents of the system, and they can brick the system if not controlled precisely and meticulously.

Magnetic attacks use magnetic fields to interfere with the local currents of the system and attempt to induce errors this way and is a very low-cost attack type.

Faults induced by attacks differ as well. Permanent faults mean the system has been changed and affected in a permanent manner. However transient faults mean that system goes back to its original characteristics with minor changes.

Attacks should be meticulous and calculated to get results desired, to get confidential information about the system. Attack fault models are use to plan an attack. On these models some assumptions can be made, types of error assumptions are described briefly as;

Bit/Byte errors are the errors that modify either bit or byte(s) of data. Specific/Random value errors are the errors that change the value of an error to a specific or totally random value. Static/Computational errors are the errors that induce errors on the memory in active use, resulting in faulty results and outputs. Data/Control errors are the errors that make the system completely bypass some of its crucial operations or functions and induces data errors.

E. Problem 5 – Reading "*Controlling PC on ARM Using Fault Injection"*

In this article we talk about vulnerabilities of the ARM 32 architecture and how to affect the Program Counter with fault injection. Most ARM devices are SOC devices. Article starts with reiterating the same information I learned from the previous article, which is various fault injection methods, I've **skipped** this part on my summary to focus this summary on new information.

On ARM Systems, some vulnerable operations like LDR and STR operations are interesting for presumed attackers because an attacker can directly control these operations, and these operations are not part of a protected portion of the controller and gives complete freedom to attacker in certain situations. In this article attacks are completely targeted at the ARM32 Load Instructions. In this case fault injection method of choice is voltage fault injection by completely taking over the power circuits of the system.

Unlike other architectures like x86 or ARM64, ARM32 architecture suffers from a major con which is how the program counter register works. Many different operations can directly affect the program counter in this architecture. Another weak point of the ARM architecture is within its secure boot sequence, which is liable to rogue flash insertions which give the attacker high privilege control over the system by injecting shellcode and pointers. This inserted code will corrupt the load functions so that they will copy the pointer values to program counter which will execute the shellcode Another attack scenario happens within Trusted Execution environment attack, which is a runtime attack. This attack utilizes the dedicated API between the REE and TEE. An attacker can take control of the REE and using the API between REE and TEE get access to TEE to access Trusted OS.

Controlled simulation tests are made within an ARM processor with ARMv7 Architecture that attempts to test the prior proposed faults. LDR Instruction is corrupted into an instruction that loads value into the Program Counter by utilizing pointers pointing to the identifier function. This corruption method differs with which register you choose to use for the corruption process. Same corruption philosophy also works for the LDMIA instruction.

Test on actual ARM Processors are conducted using the voltage fault injection method. PCB is modified to give complete control to the attackers over the power operations. Capacitors that stabilize the power delivery within the PCB are removed to ease control over the power operations. Timing of the attack is synced to a GPIO onboard to simplify it. Planned fault is injected to the process. Despite the processor running on 1.2 Volt originally, it still works within 1.1 to 1.3 volts and is more liable to faults when working at 1.1 volt. 18000 experiments are made changing the glitch parameters, VCC, Glitch pulse length, glitch delays, to gather a reference pool for the method.

Series of experiments are conducted targeting LDR and LDMIA. Pointers to a function are set on both of the instructions that prints serial interface. LDR instruction had only 1 successful glitch and %34 reset/mute rate. However, LDMIA had 27 successful glitches within the same experiment amount windows and also had %26 reset/mute rate. Results of these test tell us that LDMIA instruction is more liable to attacks than LDR instruction. However, when we analyse the successful glitches and copy their parameters and try them again, we get significantly higher success rates on both instruction attacks.

What can we do against these weak points? Countermeasures can be set at both hardware and software levels. Abusing these detected weak points depend on very tight timing windows, if the processors have random delays and varying speeds these operations become practically impossible in this fashion. Hardware protection sensors can be implemented. Processor can implement some penalties to faulty instructions to slow down the hacking process. These counter measures may not be fully effective on stopping the fault injection process.

In conclusion it is confirmed that the ARM32 architecture in fact is susceptible to the proposed LDR and LDMIA weaknesses as proved by the experiments. This poses a risk against the safety of ARM32 systems. This weakness may or may not apply to more complex architectures.

F. References

1. ARM v6-M Architecture Reference Manual, ARM