PRACTICAL ARM EXPLOITATION

LAB MANUAL

PUBLIC LAB MANUAL SAMPLE

XIPITER

AUTHORS AND INSTRUCTORS:
http://www.xipiter.com/training

Stephen A. Ridley
Stephen C. Lawler

B&D BITS & DATA ASSOCIATES

TABLE OF CONTENTS

WHAT	WHERE
Architecture Notes	4-18
The GumStix and the Lab Network	19
GDB Cheatsheet	20-21
Other command-line helpers	21
Useful IDA configurations	22-23
GDB Quirks on ARM	24
IDA Quirks on ARM	24
LABS (BASICS)	
Basics 1 (GDB on ARM)	27
Basics 1B (IDA on ARM)	30
Basics 2 (Spotting Vulns In Source) we'll probably skip this	32
Basics 3 (IDA and GDB Gangbang)	41
Basics 4 (ARM Function Calls)	44

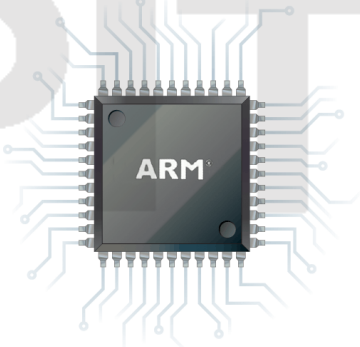
MORE CONTENT STUFF

WHAT	WHERE
Basics 5 (Shellcoding on ARM)	48
LABS (EXPLOITATION)	
Simple Stack Overflow	50
Simple Stack Overflow XN	54
Advanced Stack	57
Custom ROP (Get a RootShell!)	63
Simple Heap (Unlink exploitation)	65
Simple Heap ("Write-Me-Where")	71
MultiHeap	75
MultiHeap XN	80
MultiHeap XN & ASLR	84
APPENDIX AND OTHER STUFF	
The CPSR Register	88
Normal Glossary of Exploitation Terminology	89
Our Glossary of Exploitation Terminology	92

XIPITER

B&D BITS & DATA ASSOCIATES

SOME NOTES
&
CHEATSHEETS



ARM Architecture Reference (the basics)

- **ARM has 31 32-bit registers**
- **Only 16 of these 32 bit registers are "visible" to programmers (R0-R15):**

"VISIBLE" ARM REGISTERS		
REGISTER	NAME	DESCRIPTION
R0-R3	-	Holds function arguments. (kinda like FASTCALL on x86). R0 is also used to store the return value. Other arguments are passed on the stack. R0-R3 and R12 are considered "scratch" registers because they are not preserved across calls.
R4-R8	-	These are preserved across calls and can be used to store local variables.
R9	TR or SB	Thread Local Storage or Stack Base. This is platform dependent. Usually it is used as a general purpose register as it is on Linux.
R10	SL	Stack Limit: Sometimes used to mark the bottom edge of the stack.
R11	FP	Pointer to current frame. Like EBP on x86
R12	IP	Intra procedure stack register. Considered a "scratch" register
R13	SP	Stack Pointer (PUSH, POP, etc.)

"VISIBLE" ARM REGISTERS		
REGISTER	NAME	DESCRIPTION
R14	LR	Link Register: Holds return address for branch and link instructions.
R15	PC	Program Counter: Holds address of the next instruction to be executed (like EIP on x86).
CPSR	CPSR	Status registers (Negative, Zero, Carry, etc.) like EFLAGs on x86

- **ARM has three instruction modes:** ARM, THUMB, and Jazelle.
  - ARM is 32-bit, THUMB is 16-bit, and Jazelle is for native execution of Java
  - The instruction mode you are currently in visible in "TBIT" if CPSR register (see GDB cheatsheet and Appendix A for more info)
- **The stack grows down** the same way that it does on x86 (e.g. starts at high addresses and as frames and data are PUSHed onto the stack it grows towards lower memory addresses.)
  - During function calls, the saved PC is, by default, stored in the LR register.
  - The first four function parameters are passed by register (R0-R3), the rest on the stack: given: int foo(int a0, void *a1, char a2, int *a3, int a4, short a5); a0 through a3 would be passed in registers R0 through R3. The parameter a4 would be placed first on the stack (at SP) and a5 would be placed after it (at SP+4)
- **ARM instructions are commonly little-endian:** (For example, the 16-bit instruction 0x1337 would be stored in memory as 0x37 0x13).

- **ARM has several standard addressing modes** you should be aware of when viewing disassemblies.

ARM ADDRESSING MODES		
MODE	EXAMPLE	DESCRIPTION
Offset Addressing	[R0, 0x1337]	Access the memory at R0+0x1337
Pre-Indexed Addressing	[R0, 0x1337]!	First update R0 by adding 0x1337 to it. Then access the memory at the new R0. R0 will retain the value. This is an assignment THEN an access.
Post-Indexed Addressing	[R0], 0x1337	First access the memory at R0. Then update R0 by adding 0x1337 to it. This is an access THEN and assignment
PC Relative in IDA	=0x1337	In IDA Pro disassembly, this means that a PC-relative offset has been used to read a dword from elsewhere in memory. The value of the dword at that location is 0x1337. (IDA Pro does not indicate what the PC-relative offset is, instead it shows the actual value of the dword at that location).

FORMATS OF ADDRESSING MODE OFFSETS		
OFFSET TYPE	EXAMPLE	DESCRIPTION
Constant	MOV PC, #0x1337	0x1337 is the constant or “immediate” value.
Bitwise Offsets	R0, shift, #1	Register R0, shifted by 1 bit. shift can be one of LSL, LSR, ASR, ROR, or RRX. The RRX shift is like R0, RRX (that is, without the extra argument)

Six classes of instructions:

- Branch
- The Status Register
- Data Processing
- Load and Store
- Coprocessor
- Exception Generating

- Opcode Sizes: Fixed 32-bit instruction width. Every instruction begins at an address divisibly by four (word-aligned)

THE STATUS REGISTER:

- Conditional execution is made possible by the status register.Similar to EFLAGS on x86

CPSR				
N	Z	C	V	...
31st bit	30th bit	29th bit	28th bit	...

N: Negative Z: Zero C:Carry V:oVerflow

- **On ARM, every instruction can have a mnemonic extension** (suffix) that implies conditional statements. (e.g. “MOV” becomes “MOVEQ”, “MOVEZ”, etc)

ARM MNEMONIC EXTENSIONS		
MNEMONIC	DESCRIPTION	CPSR VALUE
EQ	Equal	Z = 1
NE	Not Equal	Z=0
CS/HS	Carry Set	C=1
CC/LD	Carry Clear	C=0
MI	Minus/Negative	N=1
PL	Plus/Positive	N=0
VS	Overflow	V=1
VC	No Overflow	V=0

More variants found on page 122 of Architecture Reference

- If flags do not satisfy condition of the instruction then instruction is considered a NOP.

```
CMP R0, #1        ;check if R0 is equal to 1
MOVNE R1, #2  ;if R0 is equal to 1 then move 2 to R1
MOVEQ R2, #3  ;else move 3 into R2
```

BRANCHING:

BRANCHING INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
B	Branch: A jump forward or backward (up to 32MB reach) like JMP on x86
BL	Subroutine call that preserves the return address into LR (R14) like CALL on x86
BX	Branch and Exchange: Uses content of a general purpose register to decide where to jump to.
BLX	Branch with Link and Exchange: A combination of BL and BX. Uses a register to decide where to jump to and then preserves return address into LR.

- Another way to change program flow is to directly change the value of PC (R15) directly (impossible on x86).

```
MOV PC, #1337 ;Redirect execution to 1337
```

DATA PROCESSING INSTRUCTIONS:

DATA PROCESSING INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
AND	Logical And (e.g. 0110 AND 1101 = 0100)
ORR	Logical inclusive Or: (e.g. 0101 OR 0011 = 0111)
EOR	Exclusive OR (XOR): (e.g. 0101 XOR 0011 = 0110)
MVN	Move Not
SUB	Subtract
ADD	Add (e.g. ADD R0, R1, R2 means R0=R1+R2)
SBC	Subtract with carry
ADC	Add with carry
TST	Test
CMP	Compare
TEQ	Test equivalence
CMN	Compare negated
MOV	Move
MUL	Multiply
CLZ	Count leading zeros
REV	Reverse the byte order plzkthnx

- **Status register transfer instructions transfer CPSR to general purpose register** and vice-versa to achieve things like:
  - Changing code conditional flags
  - Changing Processor execution mode (T-bit)
  - Changing endianness
  - Changing instruction set to Jazelle, ARM, or THUMB (more on these later)

STATUS REGISTER INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
MRS	Move status register to general purpose register
MSR	Move general purpose register to status register

```
MRS R0, CPSR                ;Read CPSR into R0
BIC R0, R0, #0xF0000000     ;Clear out N Z C and V of CPSR
MSR CPSR_f, R0              ;Move contents of R0 to CPSR. N,Z,C and V
```

LOADING AND STORING:

- To load data from a memory region into a register or store a registry value in memory there are the LDR and STR instructions:

LOADING AND STORING INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
LDR	Load a word. Ex: LDR R1, [R0] ; Loads R1 from word at the address of R0

LOADING AND STORING INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
LDRB	Load a byte. Ex: LDRB R1, [R0+4] ;Loads a byte from the address at R3+4
STR	Store a word. Ex: STR R1, [R2, #0x1337] ;Stores value in R1 to the memory address at R2+0x1337
STRB	Store a byte: Ex: STRB R1 [R2, #0x1337] ;Stores the first byte of R1 to the memory address at R2+0x1337

- The previous instructions allow you to load single registers from memory or vice versa, but ARM also allows you to load ALL (or just a few) registers from memory and vice versa.
- With Load and Store Multiple instructions the lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address.
- The instructions for loading and storing multiple are:

LOADING AND STORING MULTIPLE INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
STM	Store Multiple
LDM	Load Multiple

- Unlike other instructions, these two can not stand alone. They have required mnemonic extensions which are called “addressing mode mnemonics”. These “Addressing Mode Mnemonics” are:

MNEMONIC EXTENSIONS OF INSTRUCTIONS FOR LOADING AND STORING MULTIPLE	
MNEMONIC	DESCRIPTION
IA	Increment After
IB	Load Multiple
DA	Decrement After
DB	Decrement Before

- The following charts are an example of what happens for each Addressing Mode Mnemonic. Assume R13 has the value 0x5 (which is a unrealistic address)

INSTRUCTION	THE PLACEMENT OF REGISTERS IN MEMORY ASSUMING R13 = 0X5										
	0	1	2	3	4	5	6	7	8	9	A
STMIA R13, {R0-R1}						R0 1st byt	R0 2nd byt	R0 3rd byt	R0 4th by	R1 1st byt	R1 2nd byt
STMIB R13, {R0-R1}										R0 1st byt	R0 2nd byt

INSTRUCTION	THE PLACEMENT OF REGISTERS IN MEMORY ASSUMING R13 = 0X5										
	0	1	2	3	4	5	6	7	8	9	A
STMDA R13, {R0-R1}		**R0** 1st by	R0 2nd byt	**R0** 3rd byt	**R0** 4th byt	**R1** 1st byt	**R1** 2nd byt	**R1** 3rd byt	**R1** 4th byt		
STMDB R13, {R0-R1}	**R1** 4th byt	**R0** 1st byt	**R0** 2nd byt	**R0** 3rd byt	**R0** 4th byt						

- The previous Addressing Mode Mnemonics work for blocks of contiguous data, but what if the data you want to store or load is stack data? In this case the data must be loaded or stored in reverse order.
- For loading or storing stack data you must use the “Stack Addressing Mnemonics”:

STACK ADDRESSING MNEMONICS	
MNEMONIC	DESCRIPTION
FD	Full Descending
ED	Empty Descending
FA	Full Ascending
EA	Empty Ascending

- The following chart is an example of what happens for each Stack Addressing Mnemonic. Assume R13 has the value 0x5 (which is a unrealistic address):

INSTRUCTION	THE PLACEMENT OF REGISTERS IN MEMORY ASSUMING R13 = 0X5										
	0	1	2	3	4	5	6	7	8	9	A
STMFD R13, {R0-R1}						R1 1st byt	R1 1st byt	R1 1st byt	R1 1st byt	R0 1st byt	R0 2nd byt
STMED R13, {R0-R1}										R1 1st byt	R1 2nd byt
STMFA R13, {R0-R1}		R1 1st byt	R1 2n byt	R1 3rd byt	R1 4th byt	R0 1st byt	R0 2nd byt	R0 3rd byt	R0 4th byt		
STMEA R13, {R0-R1}	R1 4th byt	R0 1st byt	R0 2nd byt	R0 3rd byt	R0 4th byt						

- These instructions can take variable length arguments. Think of the args as a CSV.

```
STMFD R13!, {R0-R1, R5, LR}        ;Store R0-R1 and R1,R5 and LR
```

- Remember, the exclamation point (after R13 above) is “Pre-Indexed Addressing” and is used when the base register is modified (increased or decreased, depending on the addressing mode)

COPROCESSOR INSTRUCTIONS:

- ARM Architecture is “expandable” because of it’s support for co-processors.
- There are three classes of coprocessor instructions:
  - **Loading and Storing:** Instructions for transferring data to the Coprocessor via it’s Load and Store instructions.
  - **Initialization:** These instructions are used to initialize the coprocessor to begin processing data.
  - **Data Transfer:** These instructions are used to transfer values to and from coprocessor registers.
- A brief summary of coprocessor instructions:

COPROCESSOR INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
CDP	Coprocessor Data Operations
LDC	Load Coprocessor Registers
MCR	Move to Coprocessor from ARM register
MRC	Move to ARM register from coprocessor
STC	Store Coprocessor Register

- Coprocessor Instructions aren’t terribly useful to us, but it is important to at least have been introduced to them.
- During ARM execution, each coprocessor sees the same instruction set as the main ARM CPU, it simply disregards the ARM instruction set.

- If an instruction is not implemented in hardware the coprocessor throws a **Undefined Instruction** exception.

EXCEPTION GENERATING:

- There are only two instructions that throw exceptions on ARM:

EXCEPTION GENERATING INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
BKPT	Software Breakpoint (like SYSENTER on x86)
SWI	Software Interrupt (like INT3 on x86). (Sometimes shown as SVT in IDA)

- Software breakpoints trap to a debugger.
- Software interrupts allow unprivileged code (Userspace) to execute privileged code (kernel).
- As we will see on Linux based ARM we will use these SWI instructions quite a bit for making syscalls directly (like open(), bind(), execve(), etc.)

Your Lab Environment

YOUR GUMSTIX:

The Gumstix "Sand" COM is the hardware that houses our lab environment. It is slotted into an expansion development board which is how we access the ethernet ports and console/USB ports. The boards run a modified Linaro image.



- **Accessing Your Gumstix:** You will be accessing your gumstix via a wireless network SSID: "ARMExploitation". The password is "gumst!xx". You will need to SSH into the Gumstix at the IP/host provided to you by the instructor. Root password: "rootisthenewxss"

- **Filesystem:** The filesystem on the image is read-only to avoid corruption of the filesystem due to loss of power etc. Additionally, it prevents any accidental destruction of the filesystem and protects a fresh copy of the labs.

- **Scratch Area:** The read-write part of the filesystem is "/labs" you can work here. "/root/labs/" are the readonly fresh copies of the exercises.

BACKUP:

Should your hardware fail, we have additional parts that might allow us to fix it. If this does not work we have QEMU running in Amazon EC2 that can be used to complete the lab exercises. Ask us.



Debugging Cheatsheet (GDB)

GDB CHEATSHEET	
COMMAND	**DESCRIPTION**
i b	shorthand for “info breakpoints” lists all currently set breakpoints
break *0x1000	break on 0x1000
delete 1	delete breakpoint number 1
i r	shorthand for “info registers”. displays all registers and their values
set $r0=5	Set the value of register r0 to 5
stepi	step forward one instruction
bt	display a ‘backtrace’ or call stack. useful specifically after observing exceptions
nexti	step forward an instruction OVER any calls
cont	continue running
run arg1 arg2	execute the debugger target (which was set back in the shell before executing gdb) with arguments arg1 and arg2
x/10i $pc	disassemble forward 10 instructions from the value currently in PC
x/s 0x1000	display the string at memory location 0x1000
disp <expr>	do <expr> everytime a breakpoint is hit. E.G.: disp x/10i $pc

GDB CHEATSHEET	
COMMAND	DESCRIPTION
x/10xb 0x1000	display 10 bytes as hexadecimal starting at location 0x1000
x/10xw $sp	display 10 32-bit words in hexadecimal starting at $sp
set *(int*)0x8000 = 42	set 32-bits at 0x8000 to the value 42
show arm force-mode	Show the current forced instruction mode.
set arm force-mode arm	This command overrides use of the symbol table to determine whether instructions are ARM or Thumb. change “arm” to “thumb” or “auto”

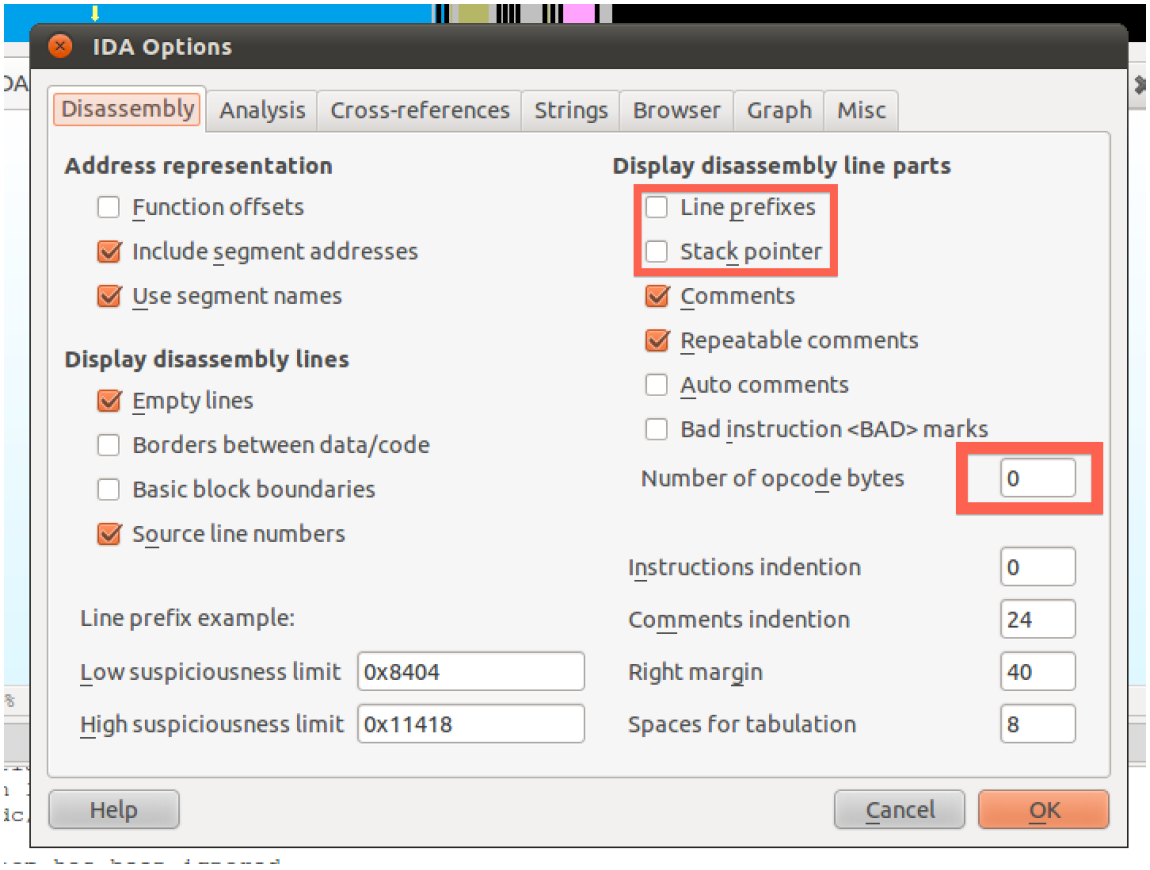
OTHER COMMAND-LINE STUFF	
COMMAND	DESCRIPTION
gdb some_file	execute “some_file” inside GDB
gdb some_file 777	attach GDB to some_file currently running in PID 777
cat /proc/777/maps	dump the address space layout of pid 777
echo 0 > /proc/sys/kernel/ randomize_va_space	Disable ASLR in the kernel (replace “0” with “1” to enable it). Will apply only to new processes. “2” adds brk aslr
execstack -s foo	Disable XN for process “foo”
ulimit -c unlimited	Turn on Core Dumps (“-c 0” turns off)
ulimit -c 0	Turn Off Core Dumps
gdb <target_app> core	open the coredump for target_app

IDA with ARM (some useful options)

IDA WITH ARM:

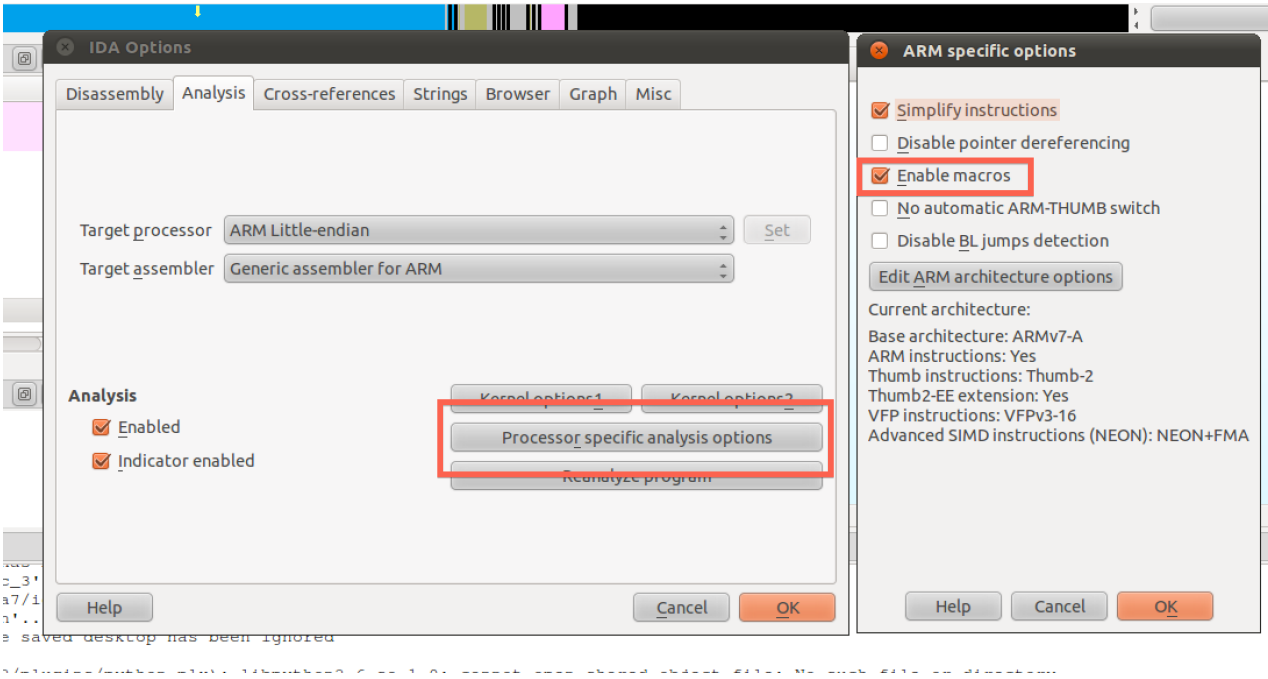
The following are some useful tidbits for using IDA on ARM binaries.

"Stack Pointer, Line Prefixes, and Opcode Bytes":



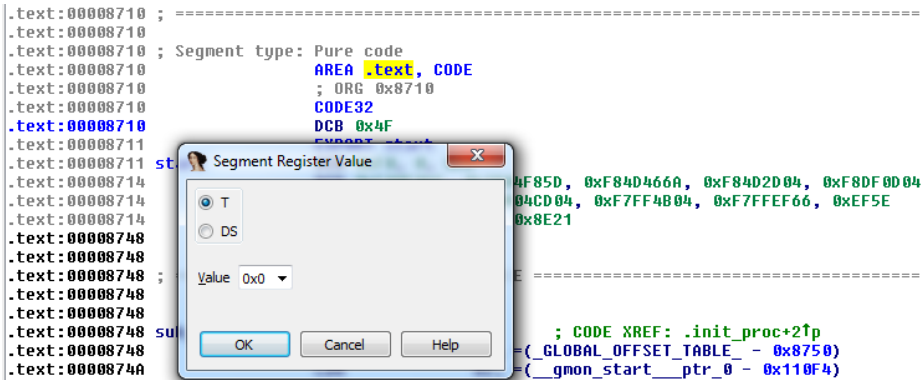
Go to "Options" then "General", click "Disassembly" tab and check these options. For "Number of Opcode Bytes" the number 3 or 6.

“Processor Specific Macros”:



Go to “Options” then “General”, click “Analysis” tab and then press “Processor Specific Analysis Options”. It will open a new dialog which will allow you to enable or disable the option.

“Switching Instruction Interpretations”:



Hit Alt-G then select the “T” radio button. Change 0x0 to 0x1 for THUMB

GDB Quirks on ARM

GDB:

GDB DISABLES ASLR!:

If you start a process under gdb (e.g., gdb ./foo as opposed to attaching an already running process gdb ./foo 2482), then gdb may actually disable ASLR in that process.

DO NOT RELY ON "disass" COMMAND:

The command "disassemble" is highly flaky due to its reliance on symbol resolution. Often the command: x/10i 0xADDRESS or x/10i $pc works more reliably for "looking ahead" in execution.

MANUALLY PATCHING ARM/LINUX BREAKPOINTS:

GDB flat out misses breakpoints sometimes. So you have to manually patch memory with an ARM elf breakpoint: **f0 01 f0 e7** or an THUMB breakpoint: **01 de.** The GDB command for this is:

1. **ARM:** set *(unsigned int*)(0xaddress) = 0xe7f001f0
2. **THUMB**: set *(unsigned short*)(0x8a74) = 0xde01
3. **THUMB2:** set *(unsigned int*)(0x8a74) = 0xa000f7f0

BREAKING BEFORE MAIN() IN AN ELF on LINUX:

There will sometimes be times that you need to break **AFTER** the ELF has been loaded but **BEFORE** main() has executed do do this you can break on "__libc_start_main" which is a function in libc that executes (for initialization reasons) before main() of the application executes.

IDA Quirks on ARM

IDA:

1. **S-WHAT!?:** IDA will sometimes display SWI instructions as SVT
2. **WHAT IS THE "DOT" SYNTAX?** IDA will sometimes display instructions as: MOV.W, LDR.W, etc. This merely indicates that the Thumb mode instruction is 32-bits.
3. **8-byte MOV instructions!? :** IDA will sometimes show things like:

   ```
   49 F2 74 70 C0 F2 06 00    MOV  R0, 0x69774
   ```

   This is actually TWO moves that IDA messes up: A 32-bit MOVW instruction (49 F2 74 70) followed by a 32-bit MOVT instruction (C0 F2 06 00) to fix this use disable "macros" as explained in the "Processor Specific Macros" settings from the previous section and
4. **Address fumbling:**

   IDA Pro seems to do a bad job of identified program offsets from regular numbers. For example I often get stuff like this:

   ```
   MOV R0, 0x4B674

   BL sub_8BA0
   ```

   0x4B674 was an obvious program address, so I had to highlight it and click Ctrl-O to get this:

   ```
   MOV R0, aHelloWorld ; "hello world!"

   BL sub_8BA0
   ```

   You get the same problems occasionally with Intel/Windows binaries in IDA Pro, it just seems more severe in ARM.

LAB EXERCISES

60+ pages cut

XIPITER

B&D BITS & DATA ASSOCIATES

APPENDIX

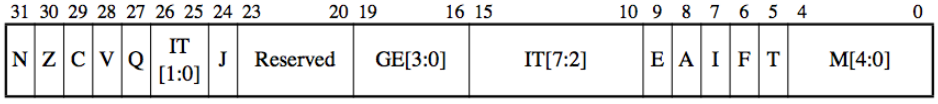
XIPITER

B&D BITS & DATA ASSOCIATES

Appendix A:Format of CPSR register

Format of the CPSR and SPSRs

The format of the CPSR and SPSRs is:



Condition code flags, bits [31:28]

Set on the result of instruction execution. The flags are:

N, bit [31] Negative condition code flag

Z, bit [30] Zero condition code flag

C, bit [29] Carry condition code flag

V, bit [28] Overflow condition code flag.

The condition code flags can be read or written in any mode, and are described in *The Application Program Status Register (APSR)* on page A2-14.

Q, bit [27] Cumulative saturation flag. This flag can be read or written in any mode, and is described in *The Application Program Status Register (APSR)* on page A2-14.

IT[7:0], bits [15:10,26:25]

If-Then execution state bits for the Thumb `IT` (If-Then) instruction. *ITSTATE* on page A2-17 describes the encoding of these bits. CPSR.IT[7:0] are the IT[7:0] bits described there. For more information, see *IT* on page A8-104.

For details of how these bits can be accessed see *Accessing the execution state bits* on page B1-18.

J, bit [24] Jazelle bit, see the description of the T bit, bit [5].

Bits [23:20] Reserved. RAZ/SBZP.

GE[3:0], bits [19:16]

Greater than or Equal flags, for SIMD instructions.

The GE[3:0] field can be read or written in any mode, and is described in *The Application Program Status Register (APSR)* on page A2-14.

E, bit [9] Endianness execution state bit. Controls the load and store endianness for data accesses:

0 Little endian operation

1 Big endian operation.

This bit is ignored by instruction fetches.

ENDIANSTATE on page A2-19 describes the encoding of this bit. CPSR.E is the ENDIANSTATE bit described there.

For details of how this bit can be accessed see *Accessing the execution state bits* on page B1-18.

GLOSSARY OF EXPLOITATION TERMINOLOGY

XIPITER

B&D BITS & DATA ASSOCIATES

Glossary

BUFFER
A region of contiguous memory (defined by its address, and a size).

BUFFER OVERFLOW OR BUFFER OVERRUN
A condition where more data has been written into a buffer than its size allows.

STACK OVERFLOW
A buffer overflow of a stack-allocated buffer (a buffer located on the stack). Rarely refers to the condition where the program attempts to allocate more stack storage than is available from the operating system.

HEAP OVERFLOW
A buffer overflow of a heap-allocated buffer (a buffer located on the heap).

BOUNCEPOINT OR GADGET
A memory address that points to a sequence of valid CPU instructions that is of use to an attacker. Sometimes referred to as a trampoline. Especially complicated bouncepoints, or when using the term return-oriented programming may be called gadgets.

RETURN-TO-LIBC OR RETURN-TO-TEXT
The concept of redirecting flow of execution back into libc (or some other text region). The basic concept is to redirect execution back to a function in libc (such as system), often in the context of a stack overflow.

STACK FLIPPING OR "PIVOTING"
The technique where an attacker uses a special sequence of opcodes to move a heap (or other) address into the stack register. This technique can, for example, be used to leverage a heap-overflowed based function pointer overwrite

into control of the stack pointer, allowing the attacker to execute a return-to-libc or ROP style attack that might not otherwise have been possible.

EXTENDED RETURN-TO-LIBC OR RETURN-ORIENTED PROGRAMMING (ROP)

A special-case of return-to-libc where an attacker returns to many different bouncepoints in libc, connected together typically via "return" statements or the logical equivalent thereof. A "return" isn't strictly necessary, only the ability to chain bouncepoints together via control-flow-transfer instructions. This may also be called branch-oriented programming, and probably someone on the Internet will invent a new term for this technique within the next 45 minutes.

MEMORY CORRUPTION

The general concept of causing the memory of a program to enter a "corrupted" or "undefined" state that could be used to construct an attack.

USE-AFTER-FREE

An attack where a buffer of memory is used by the program after it has been semantically "freed" (that is, after the point at which it should not have been used anymore).

OFF-BY-ONE

Any boundary condition that can be violated "by 1" — that is, by some small amount. Often refers to "off-by-one buffer overflows", where only a single byte of memory can be corrupted.

INTEGER OVERFLOW

A condition where integer arithmetic causes a value that is too large to store in the native word size of the CPU. Usually, the CPU will silently truncate the value to the lower 32 bits (or whatever the word size is). With exploits, this almost invariably refers to a condition where the result of this arithmetic is truncated and used to allocate a buffer that is too small to store the intended amount of memory, resulting in a buffer overflow.

GLOSSARY OF OUR EXPLOITATION TERMINOLOGY

Don'tStuffBeansUpYourNose Glossary

BUKAKHEAP
See "Heap Spray" from previous Glossary.

BOP (BRANCH ORIENTED PROGRAMMING)
An alternative to the term "ROP" or "Return Oriented Programming". Since ARM makes use of Branch instructions for the redirection of execution at the end of a function (instead of the "RET" instruction from x86) "ROP" is a bit of a misnomer from the ARM assembly perspective isn't it?

SHEAP EXPLOITATION
Make use of "Pievuts" (see below) to point the stack pointer into the heap, effectively making the heap the new location of the stack. This technique is use to evade exploitation prevention mechanisms.

SHOOKIES
Stack Cookies

CHOOKIES
Heap Chunk Cookies

LEGOS
See "Gadgets" from previous Glossary

TRICKETS
See "Gadgets" from previous Glossary

PWUNIHADS (PRONOUNCED : PUNY-HEADS)
A acronym for: People Who Use Nopsleds In Heapsprays Are Dumb

PIEVUTS:
See "Stack Flipping" or "Pivoting" in previous Glossary. An instruction sequence that allows you to pivot the Stack into the Heap as Pronounced by Georg Wicherski



FIN

XIPITER

B&D BITS & DATA ASSOCIATES