Weird Machines on Little Robots

Intro to binary exploitation on Android smartphones

@f0rki

2013-06-06

Agenda

Agenda

Introduction

- Smartphones are a Big Market
- Not as well researched as security on x86(_64)
- New challenges

on Android?

- Rooting is popular
- Increasing use of native components
  - e.g. game engines, audio/video codec stuff

Introduction

- Smartphones are a Big Market
- Not as well researched as security on x86(_64)
- New challenges

on Android?

- Rooting is popular
- Increasing use of native components
  - e.g. game engines, audio/video codec stuff

But Daddy, all the cool kids are exploiting ARM devices!!!!!!

Agenda

ARM? Embedded stuff...I think...

- Mostly sold CPU architecture
  - It's basically everywhere
- ARM Architecture is licenced to manufacturers
  - e.g. Samsung, Qualcomm, Texas Instruments, ...
  - They buy the "source code"/"blueprints" for the CPU cores
  - ...and build System-on-a-Chip (SoC)

ARM Facts

- BuzzWord Bingo:

ARM Facts

- BuzzWord Bingo:
  Bi-endian 32-Bit Load/Store RISC architecture

ARM Facts

- BuzzWord Bingo:
  Bi-endian 32-Bit Load/Store RISC architecture
  - 64-Bit on the way (AArch64)
- ARMv5 to ARMv8 are common
- (Relatively) simple architecture, no microcode
- Many extensions (like in x86 world)
- Different instruction sets
  - Fixed width instructions (32 bit or 16 bit)
  - ARM, Thumb(-2), Jazelle
  - Floating Point, SIMD instructions
  - Still R(educed)ISC?
- Power efficient

ARM Architecture and Instruction

- Registers from r0 to r15
  - r15 is Program Counter (PC)
  - r14 is Link Register
  - r13 is Stack Pointer (SP)
- Fancy features
  - conditional execution of **all** instructions
  - Bit-Shifting included (before/after instructions)
  - Several addressing modes
- ARM ABIs and ARM Procedure Call Standard (APCS)
  - Different ABI versions and sub-versions
  - ARM Embedded ABI → Android-EABI (quite similar to GNU-EABI)

Procedure Calls

- ARM has no call/ret instructions
- Direct manipulation of PC
  - ldr, pop (also: dm, ldmda, ldmdb and ldmib)
- Example Function Prologue/Epilogue

```
otherfunction:
    blx function

function:
    push {fp, lr}
    ; init stack, save registers
    ; function code
    pop {fp, pc}
```

- Arguments are passed in r0 to r4 (depending on ABI)
- Callee must preserve r4 to r8, r10, r11 and sp
  - Stack might be pretty crowded ;)

Agenda

Agenda

Exploitation 101: Science!!!

- Programs are “abstract machines” with states
- Programs transist between those states

Exploitation 101: Science!!!

- Programs are “abstract machines” with states
- Programs transist between those states
- Weird Machines
  - Program transists into undefined “weird” state
    - Through a vulnerability
    - Anything can happen (e.g. code execution)
  - State transitions still happen...

Exploitation 101: Science!!!

- Programs are “abstract machines” with states
- Programs transist between those states

- Weird Machines
  - Program transists into undefined “weird” state
    - Through a vulnerability
    - Anything can happen (e.g. code execution)
  - State transitions still happen...
  - ...and the machine gets weirder!
  - Exploitation is the art of programming of weird machines

Exploitation 101: Science!!!

- Programs are "abstract machines" with states
- Programs transist between those states

- Weird Machines
  - Program transists into undefined "weird" state
    - Through a vulnerability
    - Anything can happen (e.g. code execution)
  - State transitions still happen...
  - ...and the machine gets weirder!
  - Exploitation is the art of programming of weird machines

- Underlying problem: no distinction between code and data (von-Neumann architecture)

Exploitation is hard

- Finding vulnerabilities is hard
- Writing reliable exploits is harder
- Lot's of constraints
- Extremely architecture dependent
- Sometimes the best solution is brute-force

Agenda

Vulnerabilities I

Attack types

- Inject and execute new code (Shellcode)
- Execute existing code out of intended order (ROP)
- Data-only attacks

Buffer Overflows

The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

- Stack-based, Heap-based, in Data segment

Vulnerabilities II

Format String

- User controlled format string
- Variable arguments implementation problem
- Read arbitrary data from stack
- Write anywhere primitive using `%n`
  - Not in android libc/bionic!

Integer Overflows

- Integer values wrap around on `INT_MAX`
- Get program to increment over `INT_MAX`
- Problems with signedness ($-1 =$ `0xFFFFFFFF`)
- Usually in combination with other bugs

Vulnerabilities III

And many more. . .

Agenda

Code Execution

- Introduce your payload (shellcode or ROP “code”) into address space
- Overwrite pointer to code to your payload
  - Return address, function pointer, PLT/GOT etc.
  - Abuse linked data structures to achieve write-anywhere primitive (traditional example: heap metadata)
- Wait for usage of overwritten code pointer
- ???
- PROFIT!!!

Shellcode

- use PC-relative addressing to mix data/code
- See Phrack66/12 [1] for alphanumeric shellcodes
- Metasploit includes some Linux shellcode generators
- Use your favorite Asssembler (e.g. gcc, radare2/rasm2 [4])
- NOP-slides
  - Jump into NOP-slide
  - Reduce risk of jumping to wrong address
  - `NOP` is `mov r0, r0` (0xe1a00000)
  - Or use something other useless instead:
    e.g. `mov r1, r1` (0xe1a01001)

Return-to-lib(c)

Idea: ret2lib(c)

Prepare stack so that it looks like function call into a library on return.
(e.g. `system` function in libc)

Return-to-lib(c)

Idea: ret2lib(c)

Prepare stack so that it looks like function call into a library on return. (e.g. `system` function in libc)

BUT WAIT!

Return-to-lib(c)

Idea: ret2lib(c)

Prepare stack so that it looks like function call into a library on return. (e.g. `system` function in libc)

BUT WAIT!

- Remember: First arguments are passed in registers
- Oh noes: ret2lib(c) does not work on ARM
- We have the same Problem on `x86_64`

Return Oriented Programming (ROP)

Idea: ROP

Search for reusable code snippets that end with `ret` instruction, called gadgets. Chain together gadgets to achieve turing completeness.

Return Oriented Programming (ROP)

Idea: ROP

Search for reusable code snippets that end with `ret` instruction, called gadgets. Chain together gadgets to achieve turing completeness.

- Oh noes we have no `ret` instruction.
- Use any branching instruction!
  - Check out existing work ([5], [6])
  - Lot's of research in this area
- Though tool quality could be better

Agenda

The Android Environment

- Android is compiled with reasonably new GCC toolchain
  - Experimental support for LLVM/clang
- Userland libraries are Android specific
  - bionic as libc
  - custom linker (called “linker”)
- Many features are *inherited* by GNU/Linux

Heap protection

- malloc/free are user-space only
  - memory allocation via `brk()` systemcall
- glibc includes check to detect heap metadata tampering
- Android's bionic also includes such checks
  - in Android since 1.5
- Custom allocators might still be vulnerable
  - common in high performance code, e.g. game engines

Agenda

Stack Smashing Protection

- Stack smashing
  - stack-based buffer overflow + return address overwrite
- Prevent code execution through stack-based buffer overflows
  - Put “canary” value between return address and stack
  - Check whether canary was tampered with before returning
- Effectively mitigates stack smashing on GNU/Linux systems
  - in Android since 1.5

FORTIFY_SOURCE

- Detect (possible) buffer overflows during compile time
- Replace vulnerable functions with secure alternatives, e.g.
  - Compiler knows buffer is N bytes big
  - Replaces `strcpy(dst, src)` with `strncpy(dst, src,` N`)`
- Forces format strings to be in read-only memory
- Currently not in Android
  - Although compiler supports it
  - Missing libc support

Relocation Read Only (RELRO)

- Global Offset Table (GOT) and Procedure Linking Table (PLT)
  - Used by the dynamic linker to load shared libs
  - Contains function pointers
  - Common target for exploits
- Mark GOT/PLT as read-only if possible
  - partial – parts are still rw/not loaded yet
  - full – everything is marked ro/no lazy loading
- Support in Android linker since 4.1

Agenda

eXecute Never

- ARM supports non-executable pages
  - Bit in pagetable marks page as (non-)executable
  - Raises pagefault on instruction fetch
- Android marks stack/heap as non-executable
  - This prevents injected code from executing
- in Android since 2.3
  - Depends on the CPU
  - Most Android phones support it

eXecute Never

- ARM supports non-executable pages
  - Bit in pagetable marks page as (non-)executable
  - Raises pagefault on instruction fetch
- Android marks stack/heap as non-executable
  - This prevents injected code from executing
- in Android since 2.3
  - Depends on the CPU
  - Most Android phones support it

- Newest ARM specs include *Privileged XN*
  - Similar to Intel SMEP
  - Kernelspace (PL1) cannot fetch instructions from PXN pages
  - Userspace might still execute those pages
  - Currently not in any Device/Android

Address Space Layout Randomization I

- Randomize address space
- Attacker needs to guess addresses of i.e.
  - Address of shellcode on stack
  - Address of lib(c) for ret2lib(c)
- Makes exploits unreliable (not impossible)
- In Android since 4.0
  - Full ASLR since 4.1
  - Linker/vold was not randomized

Address Space Layout Randomization II

- Considerations
  - `fork()` preserves address layout
  - Code segment is usually not randomized (except for PIE/PIC)
  - ASLR is only effective against remote attackers
    - Android usually doesn't run many network services
    - Attackers are usually local (malicious apps)
    - Address brute-forcing is feasible on 32-bit address space
  - Info-leaks help defeat ASLR
    - Address space is the same for everything forked by zygote (all Apps)

Agenda

Exploitation Strategies

- Find non-randomized code and do ROP
  - In Java processes nearly everything is randomized
  - This makes it hard
  - Brute-Force guessing is needed
    - Unusual attack scenario
  - More luck with native binaries with big code section
- ROP to mprotect and then jump to shellcode
  - Might be easier, since we need less gadgets

Agenda

Conclusion

- Recent Android versions (> 4.1) are up to date
  - Lot's of older Android versions out there
- Android is riddled with other bugs
  - Many root exploits are based on race conditions, wrong permissions, debug stuff etc.
  - aka "Device vendors being stupid"

What next?

- Kernel
- TrustZone
- Bootloader

Go break stuff!

Agenda

References I

- Alphanumeric RISC ARM Shellcode
  Phrack Issue 66 by YYounan and PPhilippaerts
- ARM Architecture Reference Manual
  `http://infocenter.arm.com`
- ARM Procedure Call Standard
  `http://infocenter.arm.com`
- radare RE framework
  `http://radare.org/`
- Return-Oriented Programming without Returns on ARM
  Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy
  `http://www.hgi.rub.de/media/trust/veroeffentlichungen/2010/07/21/ROP-without-Returns-on-ARM.pdf`

References II

- Tim Kornau.
  Return oriented programming for the ARM architecture.
  http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf
- Exploit Mitigations in Android Jelly Bean 4.1
  https://blog.duosecurity.com/2012/07/exploit-mitigations-in-android-jelly-bean-4-1/
- A look at ASLR in Android Ice Cream Sandwich 4.0
  https://blog.duosecurity.com/2012/02/a-look-at-aslr-in-android-ice-cream-sandwich-4-0/