ADVANCED SOFTWARE EXPLOITATION ON ARM MICROPROCESSORS

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley
Stephen C. Lawler

RuxCon Breakpoint 2012

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
A bit about us...

- Former coworkers doing infosec research for a defense contractor ManTech
- Now we run a blog together, and try to work together when we can
- www.dontstuffbeansupyournose.com
Who Are We? (Ridley)

Currently: Independent Security Researcher (Xipiter)

Previously: Director of Information Security (at a bank), Senior Consultant Matasano

Senior Security Researcher McAfee (founded Security Architecture Group)

Kenshoto Founding Member, CSAW CTF Judge (Reverse Engineering)

Guest Lecturer/Instructor (New York University, Netherlands Forensics Institute, Department of Defense, Google, et al)
Who Are We? (Lawler)

Currently: Independent Security Researcher, Software Developer (Bits And Data Associates)

Previously: Principal at Mandiant, Principal at ManTech

Not originally a security guy, used to program Sonar systems for the Navy

Specializing in research, Kernel development, Kernel internals and Advanced Software Exploitation
Talk Outline

• How did we get started with this stuff?

• “Hardware Hacking for Software People” (ReCon Montreal 2011, SummerCon New York 2011)

• Diving into ARM, developing the “Practical ARM Exploitation” training.

• Building ARM exploitation development environments (emulated and “bare-metal”)

• The “Advanced exploitation techniques” we discovered for our training.
Talk Outline (cont’d)

• What the talk covers? (everything... ;-)  
  • ROP on ARM: A whole different world.  
  • Advanced Exploitation on ARM: Stack Flipping

• Conclusions/Recap
This stuff looks cool... what the hell is it?
Chips speak to each other with standard protocols!

- Simple standard serial protocols are often used!
- YOU MEAN TO TELL ME CHIPS USE SERIAL!? YES!!
- RS-232, i2c, spi, Microwire, etc
- Serial comms have low pin-counts (some as low as one wire)
- Found in: EEPROM, A2D/D2A convertors, LCDs, temperature sensors, which means EVERYTHING!
- Parallel: (hardly ever) requires 8 or more pins.
Where we found these hardware interfaces.
What Uses it?

• Analog to Digital Convertors. Found in:
  • batteries, convertors, temperature monitors

• Bus Controllers. Found in:
  • telecom, automotive, Hi-Fi systems, in your PC, consumer electronics

• Real Time Clock/Calendar. Found in:
  • telecom, consumers electronics, clocks, automotive, Hi-Fi systems, PCs, terminals

• LCD/LED Displays and Drivers. Found in:
  • telecom, automotive, metering systems, Point of Sales, handhelds, consumer electronics

• Dip Switch. Found in:
  • telecom, automotive, servers, batteries, convertors, control systems
How I’ve found it useful:

- Routers
- BlackBox Hardware PenTests
- HDMI (HDCP protocol)
- VGA (DDC/CI protocol)
- EEPROM
Our Target:

A VERY common cablemodem in the United State that uses a Broadcom chipset.
What to look at first?

Hey what are those pins?
Logs of it booting!!!

ECOS Real Time Operating System!
After fuzzing, the bugs begin to show!
Crashes!!! in the HTTP server (thttpd)

Bug in built-in HTTP server. Stack Overflow. EXPLOITED.
Now that we have crashes? What next?
Time to get good at Reverse Engineering ARM and Exploitation.
My machines are x86, where do we start with ARM?
The First Lab: QEMU
Using QEMU we got familiar with ARM:

- Got comfortable with GDB
- We got familiar with ARM architecture and idiosyncracies
- We developed our techniques and tools for writing Assembly Code and Shellcode on ARM
- We got familiar with how Interactive Disassembler (IDA) examined ARM binaries
We wrote vulnerable apps and developed our exploitation techniques

- Basic Stack Overflows
- Stack Overflows with Return-To-LibC
- Stack Overflows with “No Execute Stack” (XN)
- Advanced Stack Overflows with XN
- Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
But we wanted more...we wanted real hardware ARM!
Finding a hardware ARM Platform

- Almost every cellphone is ARM!
- Android phones are little ARM linux computers
- None of these systems are “Developer Friendly”
- We can not easily run our many tools on them:
  - languages like Lua and Python
  - shells
  - GNU Utilities, compilers, etc.
Finding a “developer friendly” hardware ARM Platform

• There are many “open” ARM platforms:
  • Raspberry Pi
  • BeagleBoard
  • ARMini
  • CuBox, etc

• We tried many many systems, and ran into many many many problems with building custom Linux distributions with adequate hardware support.
Finding a “developer friendly” hardware ARM Platform

- After a lot of trouble, we decided on GumStix platform, it met our needs the best (although slightly expensive :-)

[Image of a GumStix website showing product features and benefits]

HamiltonJet Offers Superior Propulsion Technology

Industries:
- Communications
- Education
- Energy
- Instrumentation
- Remote Data Collection
- Robotics
- Transportation

Dream, Design, Deliver: Gumstix Overo COM for development, proof of concept and production of waterjet electronic control product
Moving from emulation to “bare metal hardware” development

- Ported the exploits, shellcode, and payloads to our new hardware platform.
- Updated the Linux distribution image MANY times for “remote” access.
"Tobi" Breakout Board

Gumstix "Water" COM

The hardware

iPod Nano
The “Lackluster Hack Cluster”
Moving from emulation to “bare metal hardware” development

- We collected all of our exploitation tests and exploits into a single image we could use for reference.
The Lab Exercises
Word got out...

- Contacted by:
  - Companies that needed training on ARM exploitation
  - Companies that needed ARM reverse engineering and software exploitation work
  - many others with products (vested interest) in understanding ARM exploitation
So we did a few contracts:

- Penetration testing of many “black box devices”:
  - Smart Power Meters, “Set top boxes”, new experimental devices, new “secret” mobile devices from cellphone manufacturers
- We privately have developed techniques for exploiting software running on ARM
- Wrote exploits for all the above (Android, Windows 7 Mobile, Linux, etc)
- Developed course material to get this information out.
Developing the Course:

- Prepared our techniques so that we could publicly release them:
  - Finding new ROP gadgets on our custom ARM Linux distribution and Android.
  - Developing “user friendly” software exploitation examples.
  - Developing “Rop Library” (with examples) which includes 35+ gadgets to build payloads with.
- “Filled in the Blanks” with additional information on IDA, GDB, linking and loading, shellcoding.
What’s in our course:

• 3 to 5 Days

• 650 - 900 Slides in (15 lectures)

• 20 “Hands On” exploitation exercises on the ARM hardware

• 100 Page Lab Manual with Lab Exercise questions and detailed notes

• ARM Microprocessor Architecture Notes

• Many tools developed by us (C and Python libraries/programs) to assist with reversing and exploitation.
What our course teaches for Linux and Android

- How to reverse engineer ARM binaries with IDA (IDA bugs)
- Debugging ARM binaries with GDB
- Exploiting Stack Overflows
- Defeating Stack Overflows with “No Execute Stack” (XN)
- Exploiting Advanced Stack Overflows with XN
- Exploiting Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
- Defeating ASLR
The Course Listing

- How to reverse engineer ARM binaries with IDA (IDA bugs)
- Debugging ARM binaries with GDB
- Exploiting Stack Overflows
- Defeating Stack Overflows with “No Execute Stack” (XN)
- Exploiting Advanced Stack Overflows with XN
- Exploiting Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
- Defeating ASLR
How the course has been going:

- We are AMAZED. A course like this has never been offered.
- It sold out at Blackhat in the first two weeks.
- It SOLD OUT at CanSecWest 2012.
- It SOLD OUT at Blackhat Las Vegas 2012.
- MANY requests for private engagements of the course.
What does all this research and the popularity of our course teach us?
We are in the “Post PC” threat environment.
The world is changing...”The Post-PC Exploitation Environment”:

- Why would hackers bother with your PC when there is a GPS tracking device connected to a microphone always in your pocket?
- We trust our phones and mobile devices more than our computers and attackers know this.
- ARM Exploitation is fun and much easier than people think.
- Bugs are being found in everything from SMS messages in your iPhone to the DVR you watch Netflix on. All of these devices use ARM processors
Some Interesting Bits from the Course:
Some Interesting Bits from the Course:

ROP on ARM
(defeating XN, code-signing, et al.)
• Why bother with ROP?

  – “Execute-Never”
  – Allows virtual addresses to be marked with or without execute permission
  – If the CPU ever attempts to fetch an instruction from a virtual address without execute permission, it raises an exception (typically, delivers SIGSEGV to the offending process)
  – Therefore, an exploit must direct PC towards valid executable addresses
    • Virtual address is marked executable by the operating system
    • Address must contain valid ARM/THUMB machine code

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Why bother with ROP?

• Code-Signing
  – Some platforms verify that executable memory segments contain a valid digital signature
  – Measure is primarily a method of protecting revenue stream for application stores
  – Therefore an exploit must redirect PC to valid executable addresses

• It is not possible to have a “ret2libc” attack that calls “mprotect()” or equivalent to re-protect virtual addresses with executable page permissions
ROP: General Technique

• General technique
  – Find a number of “gadgets”
    • A few instructions, ending in an indirect branch (pop {pc}, blx r3, etc)
    • Typically, obtains values and branch targets from memory relative to SP
  – Place these gadgets, one after the other, onto the call stack
    • Such as via stack overflow vulnerability
  – The “gadget chain” will constitute a computer program (a “return-oriented” program)
  – Profit!
    • Allocate writeable, executable memory and copy shellcode into it
    • Re-protect existing virtual address space as executable and jump into it
    • Create a socket, connect out, and establish a reverse shell
    • Read contents of contacts list and send it to a remote serve via HTTP
    • Really, you can create just about any computer program by using lots of gadgets on the stack

[Website and book references]
Ret2libc, Bouncepoints, and ROP

- One of our gadgets from early in the class:
  - libc + 0x000918DC: POP {R0,R1,R2,R3,R12,LR}; BX R12
  - Loads R0-R3 with values from the stack
  - Branches to a function
  - Initializes LR to return somewhere

- On ARM, it’s really impossible to do any ret2libc without the use of a “bouncepoint” aka “gadget”
ROP: Example mprotect() call

- Goal: Use mprotect() to re-protect the stack as executable, and jump into it

<table>
<thead>
<tr>
<th>SP Offset</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>400b08dc</td>
<td>POP {R0,R1,R2,R3,R12,LR}; BX R12</td>
</tr>
<tr>
<td>00000008</td>
<td>bdffd000</td>
<td>R0: Page-aligned stack address</td>
</tr>
<tr>
<td>0000000c</td>
<td>00002000</td>
<td>R1: Length to mprotect</td>
</tr>
<tr>
<td>00000010</td>
<td>00000007</td>
<td>R2: PROT_READ</td>
</tr>
<tr>
<td>00000014</td>
<td>deadbeef</td>
<td>R3: Unused value for R3</td>
</tr>
<tr>
<td>00000018</td>
<td>400abf90</td>
<td>R12: Address of mprotect()</td>
</tr>
<tr>
<td>0000001c</td>
<td>bdffd100</td>
<td>LR: Address of the stack</td>
</tr>
</tbody>
</table>

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ROP: Example mmap() + memcpy() call

- **Goal:** Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- **Step 1:** call mmap, with that gadget that is useful for making function calls
- **Step 2:** call memcpy. It’s destination address should be the buffer we just mmap’d, it’s source address should be the contents from R6 (we know, via gdb, that R6 happens to point to our shellcode buffer at time of exploit).
- **Step 3:** jump into the buffer
ROP: Example `mmap()` + `memcpy()` call

- **Goal:** Use `mmap()` to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.

- **Step 1:** call `mmap`, with that gadget that is useful for making function calls
  - WAIT! `mmap` takes 6 arguments, not just 4
  - `mmap(addr, len, prot, flags, filedes, off)`
  - We can’t just use R0-R3 for its arguments!

- **Step 2:** call `memcpy`. ......

- **Step 3:** jump into the buffer

---

[Website and author information]

http://www.dontstuffbeansupyourownose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ROP: Example mmap() + memcpy() call

- Goal: Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- Step 1: call mmap, with that gadget that is useful for making function calls
- Step 2: call memcpy. It’s destination address should be the buffer we just mmap’d, it’s source address should be the contents from R6 (we know, via gdb, that R6 happens to point to our shellcode buffer at time of exploit).
  - WAIT! How do we “pass” R6 as the “source” address for memcpy (the 2\textsuperscript{nd} argument)? (How do we move R6 into R1? How can we do so while ensuring R0 contains the address returned by mmap?)
- Step 3: jump into the buffer
ROP: Moving R6 to R1, without changing R0

- After searching and searching, we find the following gadgets...

<table>
<thead>
<tr>
<th>Location</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc + 0x000a82d2</td>
<td>LDMIA.W R3, {R0, R1, R2, R3}</td>
</tr>
<tr>
<td></td>
<td>STMIA.W R4, {R0, R1, R2, R3}</td>
</tr>
<tr>
<td></td>
<td>B.N 0xA82A4</td>
</tr>
<tr>
<td></td>
<td>0xA82A4:</td>
</tr>
<tr>
<td></td>
<td>MOV R0, R5</td>
</tr>
<tr>
<td></td>
<td>POP {R4, R5}</td>
</tr>
<tr>
<td></td>
<td>BX LR</td>
</tr>
<tr>
<td>libc + 0x000a82d4</td>
<td>STMIA.W R4, {R0, R1, R2, R3}</td>
</tr>
<tr>
<td></td>
<td>B.N 0xA82A4</td>
</tr>
<tr>
<td></td>
<td>0xA82A4:</td>
</tr>
<tr>
<td></td>
<td>MOV R0, R5</td>
</tr>
<tr>
<td></td>
<td>POP {R4, R5}</td>
</tr>
<tr>
<td></td>
<td>BX LR</td>
</tr>
</tbody>
</table>
ROP: Moving R6 to R1, without changing R0

- After searching and searching, we find the following gadgets...

<table>
<thead>
<tr>
<th>Location</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc + 0x0001bd4c</td>
<td>MOV R0, R6</td>
</tr>
<tr>
<td></td>
<td>POP {R4, R5, R6, PC}</td>
</tr>
<tr>
<td>libc + 0x00035d1e</td>
<td>LDR LR, [SP], #4</td>
</tr>
<tr>
<td></td>
<td>ADD SP, #12</td>
</tr>
<tr>
<td></td>
<td>BX LR</td>
</tr>
<tr>
<td>libc + 0x0004c9cc</td>
<td>POP {R4, PC}</td>
</tr>
<tr>
<td>libc + 0x000b31c8</td>
<td>POP {R3, PC}</td>
</tr>
<tr>
<td>libc + 0x0001f39c</td>
<td>POP {PC}</td>
</tr>
<tr>
<td>libc + 0x000a6a40</td>
<td>MOV R3, R0; BX LR</td>
</tr>
</tbody>
</table>

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ROP: Moving R6 to R1, without changing R0

- **Step 1:** Load a good return address into LR
- **Step 2:** Load a fixed memory address ALPHA+8 into R4
- **Step 3:** Load a good return address (POP {PC}) into LR
- **Step 4:** Save R0 (mmap’d address) o the address at R4
- **Step 5:** Load a fixed memory address ALPHA into R3
- **Step 6:** Load a fixed memory address ALPHA into R4
- **Step 7:** Load/save R2 from the address at R3/R4 (effectively moving the old mmap’d address into R2)
- **Step 8:** Move R6 into R0
- **Step 9:** Load a fixed memory address ALPHA+4 into R4
- **Step 10:** Save R0 into the address at R4
- **Step 11:** Load a fixed memory address ALPHA into R3
- **Step 12:** Load a fixed memory address ALPHA into R4
- **Step 13:** Load/save R1 and R3 from the address at R3/R4
- **Step 14:** Move R3 into R0
...later that day...after much toil...
(Some time later)

```assembly
400b08dd – pop {r0-r3,r12,lr}; ...
deadbeef
00000000
00001000
00000007
00000022
400abec0 – mmap()
400af78b – add sp, #12; pop {pc}

00000000
00000000
40054d1f – ldr lr, [sp], #4; ...
4003e39d – pop {pc}
4010052c
deadbeef
4003ad4d – mov r0, r6; pop ...
deafbeef
400af78b – add sp, #12; pop {pc}
4010052c
deadbeef
40054d1f – ldr lr, [sp], #4; ...
4003e39d – pop {pc}
41414141
41414141
41414141
400c72d5 – stmia r4, ...
40100528
deadbeef
4006b9cd – pop {r4, pc}
40100530
deadbeef
400c72d5 – stmia r4, ...
40100528
deadbeef
400d21c9 – pop {r3, pc}
4010052c
deadbeef
40054d1f – ldr lr, [sp], #4; ...
4003e39d – pop {pc}
4010052c
deadbeef
40100528
deadbeef
400c72d3 – ldmia r3, ...
deadbeef
400c5a41 – mov r0, r3; pop {pc}
4005e033 – pop {r2, pc}
00000100
40075750 – memcpy()
400874bd – bx r0
```

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Uhhhh.......this is hard.

- This is getting a little complicated
- Manually stitching together “gadgets” onto the stack is error-prone and confusing
- Is there a better way?
exploit_help.py

- Python classes to make it easier to construct return-oriented programs

- 35+ ARM Linux Gadgets
  - Loading General Purpose Registers
  - Calling from registers
  - All the gadgets you need to call virtually any function with any number of arguments.
  - Students use this to build write the payloads that defeat ASLR, NX, for a full connect-back rootshell (on the last day)
exploit_help.py: Example

- **NEXT_GADGET**

  ```python
  gc = GadgetChain([  
      LOAD_AND_BRANCH_TO_LR(junk = 'A'*12),
      RET(),
      LOAD_R4(r4 = 0x40020800),
      SAVE_SCRATCH_REGS(r4 = 0xdeadbeef, r5 = 0xdeadbeef),
      NEXT_GADGET(),
      WORD(0x40020800)
  ])  
  exploit = exploit + gc.pack()
  ```
ROP on ARM Magic: “Misaligned Instructions”

• Why don’t we have “POP {R0, PC}”?

• Because NOWHERE in the entire libc binary does this instruction sequence exist. So we had to settle for “POP {R0, R2, PC}”

• But, take a look at the address of our POP {R0, R2, PC} gadget in IDA Pro…

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ARM has many instruction modes

- Recent ARM processors (e.g., ARMv7) support a number of instruction modes.
- Like most RISC architectures, ARM instructions are fixed width and must be properly aligned.
- Mode determined by the high bit of the instruction being executed. (TFlags $cpsr.t)
- This means “on the fly” mode switching! Hmm!

http://www.dontstuffbeansyouarmosc.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ARM Mode

- 32-bit instruction fixed-width and alignment
- Generally the most “featureful” of instruction modes
- Transitioned into by executing the following instructions that load the PC with the instruction set selection bit (the low order bit) cleared: BX, BLX, LDR, or LDM. As of ARMv7 this also includes: ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.
THUMB Mode

• 16-bit instruction fixed-width and alignment

• Slightly less functionality than ARM mode instructions (e.g., many 16-bit instructions can only access R0-R7)

• THUMB-2, introduced in 2003, allows for 32-bit instructions aligned on 16-bits and greater functionality when in THUMB mode

• Transitioned into by executing the following instructions that load the PC with the instruction set selection bit (the low order bit) set: BX, BLX, LDR, or LDM (aka POP). As of ARMv7 this also includes: ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.
ThumbEE Mode

- Similar to THUMB mode, but contains various extensions to support run-time generated code (JIT code)
- Transitioned into or out of via the ENTERX and LEAVEX instructions
Jazelle Mode

- Allows for native execution of Java bytecode
- Transitioned into via the BXJ instruction
ROP on ARM Magic: “Misaligned Instructions”

- I don’t see a POP {R0, R2, PC} there at all
- But wait a minute…

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ROP on ARM Magic:
“Misaligned Instructions”

• If we undefine the instruction at 3850C we see the bytes FD F7 05 BD

• What’s “05 BD” in THUMB?

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation"
ROP on ARM Magic:
“Misaligned Instructions”

- Wow, it’s POP {R0, R2, PC}!
- This is common in ROP, taking advantage of addressing offsets to create “unintended” opcode sequences

http://www.dontstuffbeansupyourtoose.com

Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Some ROP Tricks we teach: #1

- **Goal:** Read or write from scratch space
- **Problem:** We don’t know what address to use for reads/writes of memory.
- **Solution:** Just use a bukakheap’d address, or use the `.data/.bss` section of libc.
  - Specifically, the `.bss` section of libc ends at offset 0xe1528 from the start of the binary
  - But pages must be allocated as multiples of the PAGE_SIZE (4096)
  - Meaning 0xe1528 – 0xe2000 is perfect “scratch space”

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Some ROP Tricks we teach: #2

• Goal: Move the value in R2 into R1 (or R3 into R2 or R1 into R3, etc.)
• Problem: There are no gadgets to move values in volatile registers to each other.
### Some ROP Tricks we teach: #2

<table>
<thead>
<tr>
<th>Gadget Chain</th>
<th>Stack Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOAD_R4: POP {R4, PC}</strong></td>
<td>Scratch Address -&gt; R4</td>
</tr>
<tr>
<td><strong>SAVE_SCRATCH_REGS: ST Mia R4...</strong></td>
<td>SAVE_SCRATCH_REGS_BOUNCE -&gt; PC</td>
</tr>
<tr>
<td><strong>LOAD_R3: POP {R3, PC}</strong></td>
<td>Scratch Address - 4 -&gt; R3</td>
</tr>
<tr>
<td><strong>RESTORE_SCRATCH_REGS: LDMIA R3...</strong></td>
<td>RESTORE_SCRATCH_REGS -&gt; PC</td>
</tr>
</tbody>
</table>

- **Solution:**
  - Use staggered scratch address to write (for example) R2
  - And then read from that address minus 4, thereby transferring the value to R1

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Some ROP Tricks we teach: #3

• Goal: We want to write an ASCII string (or other data structure that is not merely 4 32-bit words) to somewhere in memory

• Problem: The gadget to write to memory (SAVE_SCRATCH_REGS) only works with 32-bit register values
Some ROP Tricks we teach: #3

• Goal: We want to write an ASCII string (or other data structure that is not merely 4 32-bit words) to somewhere in memory

• Problem: The gadget to write to memory (SAVE_SCRATCH_REGS) only works with 32-bit register values

• Solution: Just use SAVE_SCRATCH_REGS in exploit_help.py
Some ROP Tricks we teach: #3

<table>
<thead>
<tr>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
<th>W</th>
<th>O</th>
<th>R</th>
<th>L</th>
<th>D</th>
<th>!</th>
<th>\n</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>45</td>
<td>4C</td>
<td>4C</td>
<td>4F</td>
<td>20</td>
<td>57</td>
<td>4F</td>
<td>52</td>
<td>4C</td>
<td>44</td>
<td>21</td>
</tr>
<tr>
<td>0A</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4C4C4548
4F57204F
21444C52
0000000A

R0
R1
R2
R3

• Just visualize the data structure or string as individual byte values
• Convert those byte values to 32-bit numbers (remember, because of little-endian encoding you have to do byteswapping when representing them as numbers)
• Put the first 4 bytes into R0, as a little-endian number
• The second 4 bytes into R1, as a little-endian number
• Etc.

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Some More Interesting Bits from our Course:
ROP and Stack Overflows

• ROP – Return Oriented Programming
  – Sequence of gadgets placed on the stack
  – Takes advantage of existing opcode sequences to bypass XN or similar technology to prevent execution of stack/heap data
  – Obviously applicable in stack overflows
    • Overflow call stack with data
    • Overwrite “Saved LR” with address of your first gadget
    • Call stack contains a chain of gadgets that can be returned to, one after the other, because it was placed there by the overflow

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ROP and Heap Overflows

• ROP – Return Oriented Programming
  – Obviously applicable in heap overflows?
    • Use WWW, WMW, vtable overwrite, etc. to execute your first gadget
  • Call stack contains ... a chain of gadgets?
    – No, it won’t obviously, we are exploiting a heap overflow
    – Our chain of gadgets or ROP is on the heap somewhere
    – We have no control of the call stack at all!!
ROP and Heap Overflows

• ROP – Return Oriented Programming
  – Obviously applicable in heap overflows?
  • Use WWW, WMW, vtable overwrite, etc. to execute your first gadget
  • Call stack contains … a chain of gadgets?
    – No, it won’t obviously, we are exploiting a heap overflow
    – Our chain of gadgets or ROP is on the heap somewhere
    – We have no control of the call stack at all

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
What if there’s nothing on the stack?
THE ANSWER: PIEVUTS!

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
What if there’s nothing on the stack?

- If there is data we control on the stack we can execute ROP with a heap overflow
- What if there really is nothing on the stack?
  - Maybe we could copy data from the stack to the heap
    - For example, our bouncepoint is a gadget that copies data from R2 onto SP and then returns
    - Doable, but consider your experience with gadgets. To do something as simple as this usually requires several gadgets on the stack, and we only control one function pointer
  - Maybe we could move the address of the heap into SP and return. That is, we have to “flip” the heap into becoming the call stack
    - Back when ROP was not a publicized technique, this was called “writing an exploit”
    - Now we have a special name for it and it is called “pievutting”
ROP and Heap Overflows (when nothing’s on the stack)

Vuln calls oobj->virtual_function

Call Stack

Heap

Free Chunk(s)

VulnObject

overflow

OverwrittenObject

Free Chunk(s)

http://www.dontstuffbeansupyournos

Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
ROP and Heap Overflows (when nothing’s on the stack)

vuln calls some magical bouncepoint... and then we PWN?

Call Stack

Heap

Free Chunk(s)

VulnObject

overflow

OverwrittenObject

Free Chunk(s)

http://www.dontstuffbeansupyournosh

Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Not so fast…

• AWESOME! So we can easily PWN heap overflows now!
• But…
  – You are probably never going to find MOV SP, R0 in compiled code
  – Think about it, how often does a compiler move a register into SP?
    • Adding and subtracting to SP occurs all the time…
    • … only time you’d move a value into SP is to restore SP from a stack frame register
      • gcc (at least) almost always uses R7 for the frame register
      • Unlikely that a volatile register like R0 would ever be used for this purpose
  – What about “mis-aligned” instruction sequences?
    • Could definitely get us the MOV SP, R0
    • But, not in the libc.so binary on your QEMU VM’s…

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Flipping R7?

• R7 as frame register?
  – libc + 0x0004C652
    • MOV SP, R7; POP {R4, R5, R6, R7, R8, R9, R10, PC}
  – Restores SP from the “frame register” in R7
  – But what if the function we’ve exploited doesn’t have a frame register?
  – If it happened to store “our data” in R7, we could use this as our “pievut”

http://www.dontstuffbeansupyourownose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Flipping R7?

• Flipping R7 into SP
  – Nice, if R7 happens to point to some data we control
  – But think about it. There are FIFTEEN registers on ARM. What is the likelihood R7 points to our data?
  – We’d rather be able to use R0 as our pivot because R0 will always point to data we control (at least for vtable overwrites)
Flipping R0?

• So we scan through libc looking for “pievuts” and we eventually luck into…
  –libc + 0004f94c

• Wait what???

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Flipping R0?

• Let’s see what happens if the processor executed that instruction in ARM mode instead of THUMB…

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Flipping R0?

• Let’s spell LDMDB R0!, {R6,R12–PC} out

• It means:
  – LDMDB R0!, {R6,R12,R13,R14,PC}
  – LDMDB R0!, {R6,R12,SP,LR,PC}

• Thank goodness for ARM/THUMB mode switching!
Flipping R0?

• What does LDMDB R0!, {R6,R12–PC} do?
  – LDMDB – Load Multiple Decrement Before
  – R0 will be subtracted by 0x14 first and then registers are loaded
    • R6 loaded from original R0–0x14
    • R12 loaded from original R0–0x10
    • SP loaded from original R0–0x0C
    • LR loaded from original R0–0x08
    • PC loaded from original R0–0x04

http://www.dontstuffbeansupyournose.com
Stephen A. Ridley
Stephen C. Lawler
“Practical ARM Exploitation”
Flipping R0?

But what do we put in to SP?
What address to use?
Flipping R0?

But what do we put in to SP?
What address to use?

USE BUKAKHEAP!!!
Conclusions & Take-Aways

- The world is changing, we are entering (if not already in) a “post-pc” exploitation environment.
- ARM shellcoding and exploitation is fun! Easier that people think
- ROP on ARM actually yields many useful an interesting gadgets because of the mixed instruction modes
- NX as well as all of the modern protections on both Linux and Android can be bypassed with nuances of the ARM Microprocessor.
“Advanced Software Exploitation on ARM”

http://www.dontstuffbeansupyourownose.com

Stephen A. Ridley: @s7ephen stephen@sa7ori.org
Stephen C. Lawler: stephenlawler@bitsanddata.com

THANKS FOR LISTENING!!!!