15:05 RISC-V Shellcode

by Don A. Bailey

RISC-V is a new and exciting open source architecture developed by the RISC-V Foundation. The Foundation has released the Instruction Set Architecture open to the public, and a Privilege Architecture Model that defines how general purpose operating systems can be implemented. Even more exciting than a modern open source processing architecture is the fact that implementations of the RISC-V are available that are fully open source, such as the Berkeley Rocket Chip[7] and the PULPino.[8]

To facilitate silicon development, a new language developed at Berkeley, Chisel,[9] was developed. Chisel is an open-source hardware language built from Scala, and synthesizes Verilog. This allows fast, efficient, effective development of hardware solutions in far less time. Much of the Rocket Chip implementation was written in Chisel.

Furthermore, and perhaps most exciting of all, the RISC-V architecture is 128-bit processor ready. Its ISA already defines methodologies for implementing a 128-bit core. While there are some aspects of the design that still require definition, enough of the 128-bit architecture has been specified that Fabrice Bellard has successfully implemented a demo emulator.[10] The code he has written as a demo of the emulator is, perhaps, the first 128-bit code ever executed.

VIC® 20 OWNERS

Fulfill the expansion needs of your computer with the

RAM-SLOT MACHINE

This versatile memory and slot expansion peripheral for the Commodore Vic-20 Computer consists of a plug-in cartridge with up to 24KBytes of low power CMOS RAM and 3 additional expansion slots for ROM, RAM and I/O. The cartridge also includes a reset button (eliminates using the power-on switch) and an auto start ROM selection switch.

#RSM-8K, 8K RAM + 3 slots.......	**$ 84.50**
#RSM-16K, 16K RAM + 3 slots	**$ 99.50**
#RSM-24K, 24K RAM + 3 slots	**$119.50**

We accept checks, money order, Visa/Mastercard. Add $2.50 for shipping, an additional $2.50 for COD. Michigan residents add 4% sales tax. Personal checks—allow 10 days to clear. ® Trademark of Commodore.

K2 ELECTRONICS DESIGN CORPORATION
3990 Varsity Drive • Ann Arbor, MI 48104 • (313) 973-6266

Binary Exploitation

To compromise a RISC-V application or kernel in the traditional memory corruption manner, one must understand both the ISA and the calling convention for the architecture. In RISC-V, the term XLEN is used to denote the native integer size of the base architecture, e.g. XLEN=32 in RV32G. Each register in the processor is of XLEN length, meaning that when a register is defined in the specification, its format will persist throughout any definition of the RISC-V architecture, except for the length, which will always equate to the native integer length.

General Registers

In general, RISC-V has 32 general (or x) registers: `x0` through `x31`.[11] These registers are all of length XLEN, where bit zero is the least-significant-bit and the most-significant-bit is XLEN-1. These registers have no specific meaning without the definition of the Application Binary Interface (ABI).

The ABI defines the following naming conventions to contextualize the general registers, shown in Figure 2.[12]

[7] `git clone https://github.com/freechipsproject/rocket-chip`
[8] `http://www.pulp-platform.org/`
[9] `https://chisel.eecs.berkeley.edu/`
[10] `https://bellard.org/riscvemu/`
[11] RISC-V ISA Specification v2.1, Page 10, Figure 2.1.
[12] RISC-V ISA Specification v2.1, Page 109, Table 20.2

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired to zero	–
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	–
`x4`	`tp`	Thread pointer	–
`x5-7`	`t0-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments/return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller

Figure 2. Naming conventions for general registers according to the current ABI.

Floating-Point Registers

RISC-V also has 32 floating point registers `fp0` through `fp31`, shown in Figure 3. The bit size of these registers is not XLEN, but FLEN. FLEN refers to the native floating point size, which is defined by which floating point extensions are supported by the implementation. If the 'F' extension is supported, only 32-bit floating point is implemented, making FLEN=32.[13] If the 'D' extension is supported, 64-bit floating point numbers are supported, making FLEN=64.[14] If the 'Q' extension is supported, quad-word floating point numbers are supported, and FLEN extends to 128.[15]

Calling Convention

Like any Instruction Set Architecture (ISA), RISC-V has a standard calling convention. But, because of the RISC-V's definition across multiple architectural subclasses, there are actually three standardized calling conventions: `RVG`, Soft Floating Point, and `RV32E`.

Naming Conventions RISC-V's architecture is somewhat reminiscent of the Plan 9 architecture naming style, where each architecture is assigned a specific alphanumeric `A` through `Z` or `0` through `9`. RISC-V supports 24 architectural extensions, one for each letter of the English alphabet. The two exceptions are `G` and `X`. The `G` extension is actually a mnemonic that represents the RISC-V architecture extension set `IMAFD`, where `I` represents the base integer instruction set, `M` represents multiply/divide, `A` represents atomic instructions, `F` represents single-precision floating point, and `D` represents double-precision floating point. Thus, when one refers to `RVG`, they are indicating the RISC-V (RV) set of architecture extensions `G`, actually referring to the combination `IMAFD`.[16]

This colloquialism also implies that there is no specific architectural bit-space being singled out: all three of the 32-bit, 64-bit, and 128-bit architectures are being referenced. This is common in description of the architectural standard, software relevant to all architectures (a kernel port), or discussion about the ISA. It is more common, in development, to see the architecture described with the bit-space included in the name, e.g. RV32G, RV64G, or RV128G.

It is also worth noting here that it is defined in the specification and core register set that an implementation of RISC-V can support all three bit-spaces in a single processor, and that the state of the processor can be switched at run-time by setting the appropriate bit in the Machine ISA Register misa.[17]

Thus, in this context, the `RVG` calling convention denotes the model for linking one function to another function in any of the three RISC-V bit-spaces.

[13]RISC-V ISA Specification v2.1, Section 7.1, Page 39
[14]RISC-V ISA Specification v2.1, Section 8.1
[15]RISC-V ISA Specification v2.1, Chapter 12, Paragraph 1
[16]RISC-V Privileged Architecture Manual v1.9.1, Section 3.1.1, Page 18
[17]Ibid.
[18]RISC-V ISA Specification v2.1, Page 6, Paragraph 1

Register	ABI Name	Description	Saver
`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f10-11`	`fa0-1`	FP arguments/return values	Caller
`f12-17`	`fa2-7`	FP arguments	Caller
`f18-27`	`fs2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller

Figure 3. Floating point register naming convention according to the current ABI.

RVG RISC-V is little-endian by definition and big or bi-endian systems are considered non-standard.[18] Thus, it should be presumed that all RISC-V implementations are little-endian unless specifically stated otherwise.

To call any given function there are two instructions: Jump and Link and Jump and Link Register. These instructions take a target address and branch to it unconditionally, saving the return address in a specific register. To call a function whose address is within 1MB of the caller's address, the `jal` instruction can be used:

```
20400060:   661000ef   jal  20400ec0 <printk>
```

To call a function whose address is either generated dynamically, or is outside of the 1MB target range, the `jalr` instruction must be used:

```
204001ac:   0087a783   lw    a5,8(a5)
204001b0:   000780e7   jalr  a5
```

In both of the above examples, bits 7 through 11 of the encoded opcode equate to `0b00001`. These bits indicate the destination register where the return address is stored. In this case, 1 is equivalent to register `x1`, also known as the return address register: `ra`. In this fashion, the callee can simply perform their specific functionality and return by using the contents of the register `ra`.

Returning from a function is even simpler. In the RISC-V ABI, we learned earlier that the return address is presumed to be stored in `ra`, or, general register `x1`. To return control to the address stored in `ra`, we simply use the Jump and Link Register instruction, with one slight caveat. When returning from a function, the return address can be discarded. So, the encoded destination register for `jalr` is `x0`. We learned earlier that `x0` is hardwired to the value zero. This means that despite the return address being written to `x0`, the register will always read as the value zero, effectively discarding the return address.

THE STATE

BUILDING, SAFE DEPOSIT AND LOAN

ASSOCIATION,

OF INDIANA.

INCORPORATED UNDER THE LAWS OF THE STATE OF INDIANA.

Authorized Capital, $500,000,

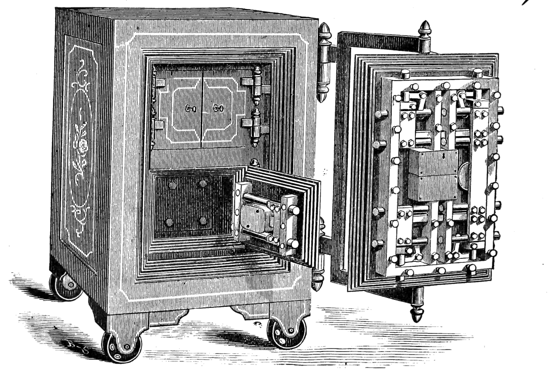
In Two Thousand Five Hundred Shares, of Two Hundred Dollars each. Monthly Payments, One Dollar and a Half per Share.

J. S. THOMPSON, PRESIDENT. H. S. SEMANS, VICE-PRESIDENT.

SAMUEL SAWYER, SECRETARY.

☞INDIANAPOLIS NATIONAL BANK, DEPOSITORY.

PETER IVEORY,



MOVER OF SAFES AND HEAVY MACHINERY

Hauling and Setting Monuments, etc. **HEAVY TRANSFER.** Orders by Telephone promptly attended to.

Office, 105 North Deleware Street, Residence, 110 Dorman Street,

INDIANAPOLIS, IND.

Thus, a return instruction is colloquially:

```
204002a8:   00008067   ret
```

Which actually equates to the instruction:

```
204002a8:   00008067   jalr  ra, zero
```

Local stack space can be allocated in a similar fashion to any modern processing environment. RISC-V's stack grows downward from higher addresses, as is common convention. Thus, to allocate space for automatics, a function simply decrements the stack pointer by whatever stack size is required.

```
20402188 <arch_main>:
20402188:   fe010113   addi  sp,sp,-32
2040218c:   80000537   lui   a0,0x80000
20402190:   80000637   lui   a2,0x80000
20402194:   00112e23   sw    ra,28(sp)

20402220:   01c12083   lw    ra,28(sp)
20402224:   02010113   addi  sp,sp,32
20402228:   00008067   ret
```

In the above example, a standard `addi` instruction (highlighted in red) is used to both create and destroy a stack frame of 32 bytes. Four of these bytes are used to store the value of `ra`. This implies that this function, `arch_main`, will make calls to other functions and will require the use of `ra`. The lines highlighted in green depict the saving and retrieval of the return address value.

This fairly standard calling convention implies that binary exploitation can be achieved, but has several caveats. Like most architectures, the return address can be overwritten in stack memory, meaning that standard stack buffer overflows can result in the control of execution. However, the return address is only stored in the stack for functions that make calls to other functions.

Leaf functions, functions that make no calls to other functions, do not store their return address on the stack. These functions, similar to other RISC architectures, must be attacked by

- Overwriting the previous function's stack frame or stored return address
- Overwriting the return address value in register ra
- Manipulating application flow by attacking a function-specific feature such as a function pointer

Soft-Float Calling Convention With regard to the threat of exploitation, the RISC-V soft-float calling convention has little effect on an attacker strategy. The `jal`/`jalr` and stack conventions from `RVG` persist. The only difference is that the floating point arguments are passed in argument registers according to their size. But, this typically has little effect on general exploitation theory and will only be abused in the event that there is an application-specific issue.

It is notable, however, that implementations with hard-float extensions may be vulnerable to memory corruption attacks. While hard-float implementations use the same `RVG` calling conventions as defined above, they use floating point registers that are used to save and restore state within the floating point ecosystem. This may provide an attacker an opportunity to affect an application in an unexpected manner if they are able to manipulate saved registers (either in the register file or on the stack).

While this is application specific and does not apply to general exploitation theory, it is interesting in that the RISC-V ABI does implement saved and temporary registers specifically for floating point functionality.

RV32E Calling Convention It's important to note the RV32E calling convention, which is slightly different from `RVG`. The `E` extension in RISC-V denotes changes in the architecture that are beneficial for 32-bit Embedded systems. One could liken this model to ARM's Cortex-M as a variant of the Cortex-A/R, except that `RVG` and `RV32E` are more tightly bound.

`RV32E` only uses 16 general registers rather than 32, and never has a hard-floating point extension. As a result, exploit developers can expect the call and local stack to vary. This is because, with the reduced number of general registers, there are less argument registers, save registers, and temporaries.

- 6 argument registers, `x10` to `x15`.
- 2 save registers, `x8` and `x9`.
- 3 temporary registers, `x5` to `x7`.

As is described earlier in this document, the general RVG model is

- 8 argument registers.
- 12 save registers.
- 7 temporary registers.

Functions defined with numbers of arguments exceeding the argument register count will pass excess arguments via the stack. In RV32E this will obviously occur two arguments sooner, requiring an adjustment to stack or frame corruption attacks. Save and temporary registers saved to stack frames may also require adjustments. This is especially true when targeting kernels.

The 'C' Extension Effect

The RISC-V C (compression) extension can be considered similar to the Thumb variant of the ARM ISA. Compression reduces instructions from 32 to 16 bits in size. For exploits where shellcode is used, or Return Oriented Programming (ROP) is required, the availability (or lack) of C will have a significant effect on the effects of an implant.

An interesting side effect of the C extension is that not all instructions are compressed. In fact, in the Harvest OS kernel (a Lab Mouse Security proprietary operating system), the compression extension currently only results in approximately 60% of instructions compressed to 16 bits.

Because the processor must evaluate the type of an instruction at every fetch (compressed or not) when compression is available, there is a CISC-like effect for exploitation. Valid compressed instructions may be encoded in the lower 16 bits of an existing 32-bit instruction. This means that someone, for example, implementing a ROP attack against a target may be able to find useful 16 bit opcodes embedded in intentional 32-bit opcodes. This is similar to a paper I wrote in 2002 that demonstrated that ROP on CISC architectures (then called return-to-text) could abuse long multi-byte opcodes to target useful bytes that represented beneficial opcodes not intended to be used by the compiler.[19]

```
20400032 <lock_unlock>:
20400032: 0a05202f  amoswap.w.rl  zero,zero,(a0)
20400036: 4505      li        a0,1
20400038: 8082
```

Since the C extension is not a part of the RVG IMAFD extension set, it is currently unknown whether C will become a commonly implemented extension. Until RISC-V is more predominant and a key player arises in chip manufacturing, exploit developers should either target their payloads for specific machines, or should focus on the uncompressed instruction set.

Observations

Exploitation really isn't so different from other RISC targets, such as ARM. Just like ARM, the compression extension isn't necessary for ROP, but it can be handy for unintentionally encoded gadgets. While mitigations like `-fstack-protection[-all]` are supported, they require `__stack_chk_{guard-,fail}`, which might be lacking on your target platform. For Linux targets, be sure to enable `PIE, now, relro` for ASLR and GOT hardening.

Building Shellcode

Building shellcode for any given architecture generally only requires understanding how to satisfy the following abstractions:

- Allocating memory.
- Locating static data.
- Calling routines.
- Returning from routines.

Allocating Memory

Allocating memory in RISC-V environments is similar to almost any other processing environment for conventional operating systems. Since there is a stack pointer register (sp/x2), the programmer can simply take a chance and allocate memory via the stack. This presumes that there is enough available memory in the system, and that a fault won't occur. If the exploitation target is a userland application in a typical operating system, this is always a reasonable gamble as even if allocating stack would fault, the underlying OS will generally allocate another page for the userland application. So, since the stack grows down, the programmer only needs to decrement the sp (round up to a multiple of 4 bytes) to create more space using system stack.

[19]Sendmail Prescan Exploitation and CISCO Encodings (127 Research & Development, 2002)

Some environments may allocate thread-specific storage, accessible through a structure stored in the thread pointer (`tp`/`x4`). In this case, simply dereference the structure pointed to by `x4`, and find the pointer that references thread-local storage (TLS). It's best to store the pointer to TLS in a temporary register (or even `sp`), to make it easier to abuse.

As with most programming environments, dynamic memory is typically also available, but must be acquired through normal calling conventions. The underlying mechanism is usually `malloc`, `mmap`, or an analog of these functions.

Locating Static Data

Data stored within shellcode must be referenced as an offset to the shellcode payload. This is another normal shellcode construct. Again, RISC-V is similar to any other processing environment in this context. The easiest way to identify the address of data in a payload is to find the address in memory of the payload, or to write assembly code that references data at position independent offsets. The latter is my preferred method of writing shellcode, as it makes the most engineering sense. But, if you prefer to build address offsets within executable images, the usual shellcode self-calling convention works fine:

```
0000000000000000 <lol>:
 0:   0100006f   j      10 <bounce>
0000000000000004 <lol2>:
 4:   00000513   li     a0,0
 8:   0000a583   lw     a1,0(ra)
 c:   00000073   ecall
0000000000000010 <bounce>:
10:   ff5ff0ef   jal    ra,4 <lol2>
0000000000000014 <data>:
14:   0304       addi   s1,sp,384
16:   0102       slli   sp,sp,0x0
```

As you can see in the above code example, the first instruction performs a jump to the last instruction prior to static data. The last instruction is a jump-and-link instruction, which places the return address in `ra`. The return address, being the next instruction after jump-and-link, is the exact address in memory of the static data. This means that we can now reference chunks of that data as an offset of the `ra` register, as seen in the load-word instruction above at address `0x08`, which loads the value `0x01020304` into register a1.

It's notable, at this point, to make a comment about shellcode development in general. Artists generally write raw assembly code to build payloads, because it's more elegant and it results in a much more efficient application. This is my personal preference, because it's a demonstration of one's connection to the code, itself. However, it's largely unnecessary. In modern environments, many targets are 64-bit and contain enough RAM to inject large payloads containing encrypted blobs. As a result, one can even write position independent code (PIC) applications in C (and even C++, if one dares). The resultant binary image can be injected as its own complete payload, and it runs perfectly well.

But, for constrained targets with little usable scratch memory, primary loaders, or adversaries with an artistic temperament, assembly will always be the favorite tool of trade.

Calling Routines

Earlier in this document, I described the general RISC-V calling convention. Arguments are placed in the `aN` registers, with the first argument at a0, second at a1, and so-forth. Branching to another routine can be done with the jump-and-link (`jal`) instruction, or with the jump-and-link register (`jalr`) instruction. The latter instruction has the absolute address of the target routine stored in the register encoded into the instruction, which is a normal RISC convention. This will be the case for any application routine called by your shellcode.

The Linux syscall convention, in the context of RISC-V, is likely similar to other general purpose operating systems running on RISC-V processors. The Linux model deviates from the generic calling convention by using the `ecall` instruction. This instruction, when executed from userland, initiates a trap into a higher level of privilege. This trap is processed as, of course, a system call, which allows the kernel running at the higher layer of privilege to process the request appropriately.

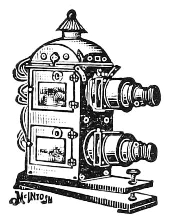
System call numbers are encoded into register `a7`. Other arguments are encoded in the standard fashion, in registers `a0` through `a6`. System calls exceeding seven arguments are stored on the stack prior to the call. This convention is also true of general routine calls whose argument totals exceed available argument registers.

Returning from Routines

Passing arguments back from a routine is simple, and is, again, similar to any other conventional processing environment. Arguments are passed back in the argument register `a0`. Or, in the argument pair `a0` and `a1`, depending on the context.

This is also true of system calls triggered by the `ecall` instruction. Values passed back from a higher layer of privilege will be encoded into the `a0` register (or `a0` and `a1`). The caller should retrieve values from this register (or pair) and treat the value properly, depending on the routine's context.

One notable feature of RISC-V is its compare-and-branch methodology. Branching can be accomplished by encoding a comparison of registers, like other RISC architectures. However, in RISC-V, two specific registers can be compared along with a target in the event that the comparison is equivalent. This allows very streamlined evaluation of values. For example, when the standard system call `mmap` returns a value to its caller, the caller can check for `mmap` failure by comparing `a0` to the `zero` register and using the branch-less-than instruction. Thus, the programmer doesn't actually need multiple instructions to effect the correct comparison and branch code block; a single instruction is all that is required.



Amusement
Entertainment
Instruction

MONEY MADE EASILY.
REQUIRES BUT SMALL INVESTMENT.

STEREOPTICONS

Accessory Apparatus, Lantern Slides.

Write for Catalogue. Mention McClure's.

McINTOSH BATTERY & OPTICAL CO., Chicago.

Putting it Together

The following example performs all actions described in previous sections. It allocates 80 bytes of memory on the stack, room for ten 64-bit words. It then uses the aforementioned bounce method to acquire the address of the static data stored in the payload. The system call for socket is then called by loading the arguments appropriately.

After the system call is issued, the return value is evaluated. If the socket call failed, and a negative value was returned, the `_open_a_socket` function is looped over.

If the socket call does succeed, which it likely will, the application will crash itself by calling a (presumably) non-existent function at virtual address `0x00000000`.

As an example, the byte stored in static memory is loaded as part of the system call, only to demonstrate the ability to load code at specific offsets.

```
0000000000000000 <lol>:
 0:   fb010113   addi   sp,sp,-80
 4:   00113023   sd     ra,0(sp)
 8:   00813423   sd     s0,8(sp)
 c:   0200006f   j      2c <bounce>
0000000000000010 <_open_a_socket>:
10:   00200513   li     a0,2
14:   00100593   li     a1,1
18:   00600613   li     a2,6
1c:   00008883   lb     a7,0(ra)
20:   00000073   ecall
0000000000000024 <_crash_or_loop>:
24:   fe0546e3   bltz   a0,10 <_open_a_socket>
0000000000000028 <_crash>:
28:   00000067   jr     zero
000000000000002c <bounce>:
2c:   fe5ff0ef   jal    ra,10 <_open_a_socket>
0000000000000030 <data>:
30:   00c6       slli   ra,ra,0x11
```

Big shout out to `#plan9` for still existing after 17 years, TheNewSh for always rocking the mic, Travis Goodspeed for leading the modern zine revolution, RMinnich for being an excellent resource over the past decade, RPike for being an excellent role model, and my baby Pierce, for being my inspiration.

Source code and shellcode for this article are available attached to this PDF and through Github.[20]

[20] `git clone https://github.com/donbmouse/riscv-security || unzip pocorgtfo15.pdf riscv-security.zip`