Execution at RISC: Stealth JOP Attacks on RISC-V Applications

Loïc Buckwell[0009−0001−4848−1478], Olivier Gilles[0000−0002−3776−2071], Daniel Gracia Pérez[0000−0002−5364−8244], and Nikolai Kosmatov[0000−0003−1557−2813]

Thales Research & Technology, Palaiseau, France
{loic.buckwell,olivier.gilles,daniel.gracia-perez, nikolai.kosmatov}@thalesgroup.com

Abstract. RISC-V is a recently developed open instruction set architecture gaining a lot of attention. To improve the security of these systems and design efficient countermeasures, a better understanding of vulnerabilities to novel and future attacks is mandatory. This paper demonstrates that RISC-V is sensible to Jump-Oriented Programming, a class of complex code-reuse attacks. We provide an analysis of new dispatcher gadgets we discovered, and show how they can be used together to build a stealth attack, bypassing existing protections. We implemented a proof-of-concept attack on an embedded web server compiled for RISC-V, in which we introduced a vulnerability allowing an attacker to read an arbitrary file from the remote host machine.

Keywords: Control-Flow Integrity · Code-Reuse Attacks · Embedded Systems · RISC-V.

1 Introduction

The RISC-V Instruction Set Architecture (ISA)[1] is a novel open Reduced Instruction Set Computer (RISC) ISA, which is often used for embedded systems. While RISC ISAs innately have a smaller attack surface than Complex Instruction Set Computer (CISC) ISAs, many of them run critical systems, including industrial control systems or cyber-physical systems, whose failure may have dramatic consequences (environmental disasters, loss of human lives...). Using a novel open ISA has several benefits. Its novelty brings security advantages by taking past failures into experience. Even more important is its open status, as trust in the architecture relies on community review. This also enables national independence in microchip supplies; a very important feature as target systems may be strategical and export restrictions become more common.

While most RISC-V architectures allow a satisfying level of security compared to similar classes of systems [15], they will increasingly become the target to complex attacks as their relevance in the industrial and strategical field increases. Eventually, state-backed attackers are bound to attack them. In order

[1] https://riscv.org

to anticipate this threat, security researchers face the challenge to find potential vulnerabilities and imagine suitable protection mechanisms. Code-Reuse Attacks (CRA), and specifically Jump-Oriented Programming (JOP), are among the most complex attacks to realize, but also to prevent. They can be very powerful when successful, as they can allow the attacker to run an arbitrary sequence of instructions within the corrupted application. In this article we adopt the attacker's point of view and try to perform a JOP attack, with the intent of (1) getting a better understanding of RISC-V systems vulnerabilities, and (2) ultimately designing better countermeasures to prevent these attacks.

Contributions. We summarize our contributions as follows:

- a first analysis of vulnerabilities to JOP attacks on the RISC-V architecture;
- a description of new dispatcher gadgets enabling JOP attacks to bypass modern mitigations on the RISC-V architecture while increasing its attack surface;
- a demonstration of feasibility by implementing and testing a stealth JOP attack on a vulnerable RISC-V application.

Outline. Section 2 introduces code-reuse attacks, countermeasures against them and the limitations of the latter. Section 3 introduces a new kind of dispatcher gadget we found, increasing functional gadgets availability to the level of ROP attacks. Section 4 describes a stealth attack we developed against a vulnerable RISC-V application using techniques described in previous sections. Section 5 compares our approach to other efforts related to Jump-Oriented Programming and RISC-V security. Finally, Section 6 provides a conclusion.

2 Code-Reuse Attacks Overview

The aim of a *Code-Reuse Attack* (CRA) — in opposition to code injection attacks — is to reuse existing code in a target application in order to perform unintended and often malicious actions. It is not in itself a vulnerability, but relies on an earlier memory corruption allowing to hijack the execution flow. Such vulnerabilities are well-known, but still prevalent in many systems [23]. An example of a CRA is return-to-libc [21], where the execution flow is redirected to a single function after manipulation of arguments within the stack of the corrupted function. More sophisticated attacks with the same principle of stack corruption have emerged, among which the *Return-Oriented Programming* (ROP) technique [19,4]. It consists in chaining *gadgets*, i.e. code snippets composed of a few instructions and ending with a linking instruction. In the case of ROP, the linking instructions are "return to caller" which pop and jump to the next gadgets' addresses stored in the corrupted stack. Using this approach, the attacker can run an arbitrary sequence of legit instructions.

2.1 Countermeasures

Multiple methods were proposed and used in order to defend against return-to-libc and ROP. Address Space Layout Randomization (ASLR) randomizes base

addresses of memory mappings. Stackguard [6] introduced the notion of canaries to protect the integrity of the stack. Yet solutions relying on secrets depend much on the system entropy, which tends to decline as the system uptime increases — an important issue for embedded systems that can run for decades without reboot. Both ASLR and Stackguard are even weaker on 32-bit systems [20], and several techniques have been proposed to bypass them.

Abadi et al. [1] first formally identified a process property named *Control-Flow Integrity* (CFI), defined by the adherence of the runtime execution flow to its intended behavior. In order to ensure this property against attackers, they proposed two complementary protections: shadow stack and landing pads[2]. *Shadow stack* protects backward-edge jumps by pushing procedure return addresses to a memory protected stack at call time. When a procedure returns, its return address is popped from both stacks and compared. If they differ, a memory corruption is detected. *Landing pads* are special instructions protecting forward-edge jumps. When implemented, each jump destination must be one of these instructions. However, even if an application is compiled with landing pads, it can still use shared libraries that are not, effectively loosing benefits for the corresponding code. Nevertheless, this protection makes theoretically all kinds of CRA nearly impossible to implement, and do indeed stop most return-to-libc and ROP attacks, although often leading to significant fall of performances [3], as opposed to shadow stacks which can be efficiently implemented in hardware, particularly in systems with limited dynamicity such as many embedded and/or critical systems. Hence, landing pads are less likely to be fully implemented in these systems.

2.2 Jump-Oriented Programming

Much like ROP, JOP consists in assembling *functional gadgets* containing useful instructions present in the target application in order to perform a malicious action. However JOP attacks do not rely on a corrupted stack: the chaining mechanism is done by a *dispatcher gadget*. Its role is to load (see ⓑ in Figure 1) and jump (cf. ⓒ) to the next functional gadget from a *dispatch table*, generally injected into a buffer. Each functional gadget must then end with a jump to the dispatcher gadget (cf. ⓓ). To do this, at least two registers need to be reserved: one for the dispatcher gadget (*dispatcher gadget register*) and one for the dispatch table (*dispatch table register*). The *initializer gadget* is responsible to set these registers and to pass control to the dispatcher gadget (cf. ⓐ). Figure 1 illustrates this mechanism with an example, where `s1` and `a5` are reserved registers (respectively, for the dispatch table and the dispatcher gadget), and `a6` is used to branch to functional gadgets. In this example, the initializer gadget sets a reserved register from the current stack frame, assuming it is under the attacker's control.

[2] A specification of shadow stack and landing pads for RISC-V is currently under ratification, see `https://github.com/riscv/riscv-cfi/`.

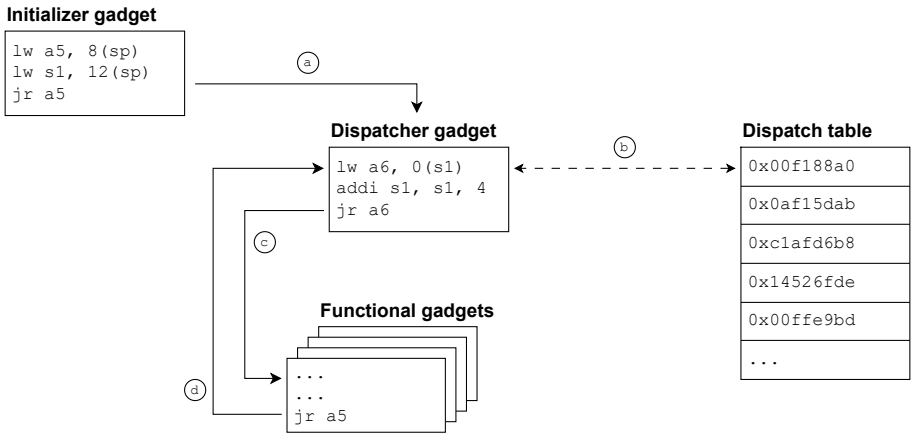

Fig. 1. JOP mechanic principle.

Building a JOP attack is far more complex than building a ROP attack for several reasons. First, initializer and dispatcher gadgets operating on the same registers need to be found to maintain the JOP chain. Although patterns leading to these gadgets are quite simple, there are very few of them in practice, and viable combinations of both gadgets are even more scarce. In addition to this difficulty, for any viable pair, there must be enough compatible functional gadgets in order to build the actual attack code (i.e. gadgets ending with a jump to the dispatcher gadget register). However, the vast majority of functional gadgets are procedure returns[3] but using them would trigger shadow stack detection if there is one. Argument registers are also a bad option for the dispatcher gadget register as it would prevent argument passing in the JOP chain. Other registers must be used but are less common, reducing the attack surface. Last but not least, side-effects in functional gadgets must also be considered as they can break the gadget chain management by clobbering the reserved registers.

Table 1 shows the number of available gadgets per register in the GNU libc 2.34 compiled for RISC-V 32 bits with M, A and C extensions (RV32IMAC). These statistics has been gathered with RaccoonV[4], an open-source tool we developed to find RISC-V JOP gadgets.

Register	`ra`	`a5`	`t1`	`t3`	`tp`	`a4`	`s0`	`s2`	`a2`	`a0`	`sp`	`s1`	`a3`	`t5`	`s8`
Available gadgets	4557	810	318	255	239	184	183	157	147	106	97	86	83	79	68

Table 1. Gadget availability per register in libc (top 15).

For these reasons, JOP attacks remain mostly theoretical. To our knowledge, there is no publicly known example of a JOP attack. In the following sections, we demonstrate the feasibility of JOP attacks on applications compiled for the

[3] For the RISC-V architecture, it corresponds to gadgets ending with a jump to `ra`.
[4] `https://github.com/lfalkau/raccoonv`.

RISC-V architecture, and introduce a new kind of dispatcher gadgets enabling the use of procedure epilogues as functional gadgets without triggering shadow stack detection, allowing to craft stealth attacks with a greater attack surface.

3 Autonomous Dispatcher Gadget

In a previous work [10], we managed to build a JOP attack on a RISC-V application but because of the aforementioned limitations, we did not succeed to chain several syscalls, limiting our results. From this experience, we started to search for more suitable dispatcher gadgets, and found a new kind of them that we called *Autonomous Dispatcher Gadget* (ADG). Figure 2 shows an ADG example and illustrates the JOP principle when using it.

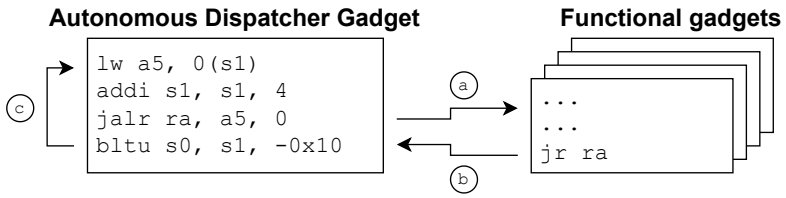

Fig. 2. JOP mechanism with an autonomous dispatcher gadget.

Unlike classic dispatcher gadgets, an ADG links to functional gadgets through a JALR instruction, which stores the next instruction's address in its first operand register (`ra`) and jumps to the target register (see ⓐ in Figure 2). Regarding the specification, this is a procedure call. For this reason, functional gadgets ending in a jump to `ra` (procedure returns) can — and must — now be used in order to avoid shadow stack detection. When they return, control is given back to the saved return address, which is the instruction right after the JALR in the ADG (cf. ⓑ). This brings us to the second key point that makes the ADG's mechanic suitable: the instruction right after the JALR is a branching instruction that self-links the ADG to itself (cf. ⓒ). From our experience, this branching instruction is always conditional, but as we will show in Section 4, ensuring the condition remains true is quite easy.

Reserved registers. In order to craft a JOP attack using an ADG, only one reserved register is required: the dispatch table register. The dispatcher gadget register is no longer required as the ADG links back to itself with a branching instruction. As a consequence, the initializer gadget can use any register to jump to the dispatcher gadget the first time (except `ra` — this would trigger shadow stack detection). Reducing the reserved registers constraint between these two gadgets considerably increases the probability to find a compatible pair. Moreover, it also increases the amount of available functional gadgets as only one register is to be preserved from side-effects in order to avoid breaking the chain.

Code pattern. The first autonomous dispatcher we found was in the GNU libc 2.34, compiled with the second GCC level of optimization (`-O2`). It seems to be located in the `__call_tls_dtors` function (which runs the destructors in sequence when the program exits), but we cannot confirm it as the library is stripped on our system. We also managed to reproduce several ADGs with simple and realistic code patterns, each involving function pointer calls inside a loop. Figure 3 shows one of them, and the corresponding generated ADG.

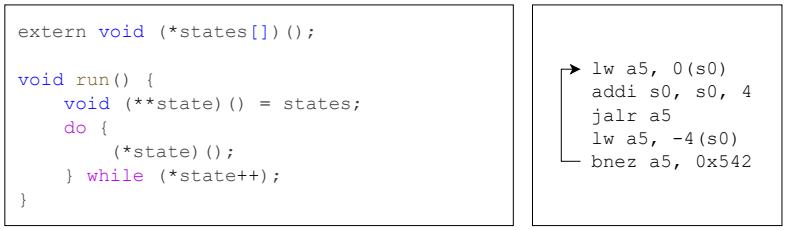

Fig. 3. ADG code pattern and resulting gadget.

From our experience, generated ADGs often use the first available saved registers (`s0-11`) as the dispatch table register, which is convenient because there is a good balance of gadgets loading them from the stack (potential initializer gadgets) and gadgets which do not clobber them, making them compatible functional gadgets. In a similar way, the first available argument register starting from `a5` seems to be used by GCC to hold function pointers. This is also convenient as it allows an attacker to pass at least 4 arguments between functional gadgets to perform syscalls.

4 Attacking Real-World RISC-V Applications

In order to prove the feasibility of JOP on the RISC-V architecture using an ADG, we implemented a proof-of-concept attack against a well-known embedded application: the Mongoose web server; and more precisely their provided http-server, in which we introduced a memory corruption vulnerability. Mongoose is a target of choice because it exposes a remote service on the network and is widely available in many embedded products for configuration purpose.

In this section, we show how we were able to remotely read the root's private SSH key stored on disk with a JOP exploit by crafting a malicious HTTP request, thanks to an ADG.

Attack model. Mongoose is developed in C. We compiled it for a Linux RISC-V 32-bit system using GCC. The binary and linked libraries have been compiled with modern protections, using `-fstack-protector-all -D_FORTIFY_SOURCE=2`. The second level of optimization (`-O2`) has also been used for libraries. However, we disabled ASLR on the target system since it can be bypassed by different

techniques [11] [9], and is out of scope of this research. Position Independent Code (PIE) has also been disabled on the target binary. Both operating system and target application were executed and validated on a RISC-V CVA6 32-bit softcore design[5] [24] running Linux, deployed on a Genesys2 FPGA.

We also made the hypothesis that the attacker is able to access an exact twin of the target application (either by rebuilding it with the same options and environment, or by acquiring a device running said application), so static analysis can be performed both on target binary and linked libraries.

Attack vector. As a first step the attacker must identify a memory vulnerability allowing to hijack the execution flow toward an initializer gadget. In our experiment, we introduced a format string vulnerability within the HTTP request handler of the target application: the body of the HTTP request is passed as the first printf argument, which permits an attacker to perform arbitrary write operations. Figure 4 shows the diff of the introduced vulnerability.

```
diff --git a/src/mongoose/main.c b/src/mongoose/main.c
index 1399f3c..0febf84 100644
--- a/src/mongoose/main.c
+++ b/src/mongoose/main.c
@@ -29,9 +29,10 @@ static void cb(struct mg_connection *c, int ev, void *ev_data, void *fn_data) {
                mg_http_parse((char *) c->send.buf, c->send.len, &tmp);
                cl = mg_http_get_header(&tmp, "Content-Length");
                if (cl == NULL) cl = &unknown;
-               MG_INFO(("0[%.*s] 2[%.*s] 3[%.*s] 4[%.*s]", (int) hm->method.len, hm->method.ptr,
-                        (int) hm->uri.len, hm->uri.ptr, (int) tmp.uri.len, tmp.uri.ptr,
-                        (int) cl->len, cl->ptr));
+               char body[16384] = {0};
+               sprintf(body, "%.*s", hm->body.len, hm->body.ptr);
+               printf(body);
+               fflush(0);
        }
        (void) fn_data;
 }
```

Fig. 4. Mongoose introduced vulnerability diff.

Gadgets research. The identification of available gadgets in the application is very important as it decides which assets can be targeted by the attack. Too few, or not diverse enough gadgets will reduce the attack surface in the best case, or make the attack impossible in the worst case.

We decided to exclusively use gadgets we found in the GNU C library (libc), as (1) it contains a lot of code and supposedly offers a great amount of gadgets and (2) almost all binaries are linked to it, making the attack more portable.

To identify gadgets, we used RaccoonV, which accepts queries to find gadgets based on their characteristics.

Figure 5 shows a simple query output, where we searched for gadgets in the libc, that loads the immediate 0 (`--op=li --imm=0`) in the register `a2` (`--rr=a2`), and is at most 1 (`--max=1`) instruction long (excluding the final linking instruction).

[5] `https://github.com/openhwgroup/cva6`.

```
$ rv libc.so.6 --op=li --wr=a2 --imm=0 --max=1
0x0006f082      01 46   li a2, 0
0x0006f084      82 80   jr ra

0x000d4244      01 46   li a2, 0
0x000d4246      02 94   jalr s0

0x000a1fbc      01 46   li a2, 0
0x000a1fbe      82 97   jalr a5

----------
Found 3 unique gadgets.
```

Fig. 5. RaccoonV output with a query on libc.

As the dispatcher gadget is among the hardest to find, it is strongly advised to find one first and to build the attack around. As said previously in Section 3 we found an ADG, shown in Figure 6. Its self-linking instruction is conditional, and in order to use it without breaking the chain, we must ensure `s0` remains inferior than `s1`.

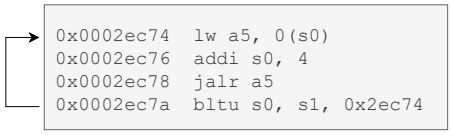

Fig. 6. Autonomous dispatcher gadget found in libc.

Finding an initializer gadget is not an easy task either but thanks to the ADG, we had less constraints on loaded registers and managed to find the one illustrated in Figure 7. While it contains some side-effects we will need to handle later (stack pointer increment), it allows us to load both `s0`, `s1` and `a5` from the current stack frame before jumping to `r5`.

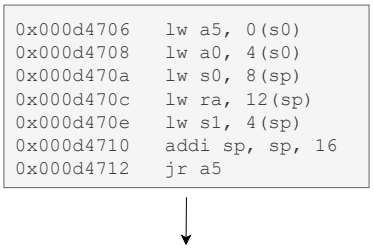

Fig. 7. Initializer gadget found in libc.

Using these two gadgets together to build our JOP attack allows using functional gadgets ending in `ra` without triggering shadow stack detection, bringing the attack surface to the same level than ROP attacks.

Definition of the attack objective. The objective of the attack is to be decided from the number of available functional gadgets identified. Having a large number of compatible functional gadgets (4557 in our case) gives the attacker enough freedom to build complex attack code, as long as it does not involve control-flow operations.

In our experiment, the objective is to read the root's private SSH key file used to administrate the server without being detected, and to return to original code, so that the web server still runs fine afterwards. We define the attack code as equivalent to the C code shown in Figure 8.

```
void attack_code() {
   int fd = openat(0, "/home/root/.ssh/id_rsa", O_RDONLY);
   read(fd, buf, 3000);
   write(5, buf, 3000);
}
```

Fig. 8. C formalization of the attack objective.

The first `openat` argument is unused when the path is absolute so we can ignore it. We used 5 as the file descriptor to write the key as it turned to be the first file descriptor assigned to clients by the HTTP server. If the file descriptor is free when the attacker sends the malicious request, the communication socket will use it; otherwise, the attack will fail, but can be retried anytime later (or immediately, setting the first write argument to another value).

JOP chain design. Once the objective of the attack is defined, the actual gadget chain can be crafted. In our case, it consists in a sequence of 3 syscalls and a cleanup step allowing to return to the nominal application code without crashing. Using RaccoonV, we managed to build the gadget chain shown in Figure 9, where "..." represents instructions that are not useful to understand the attack.

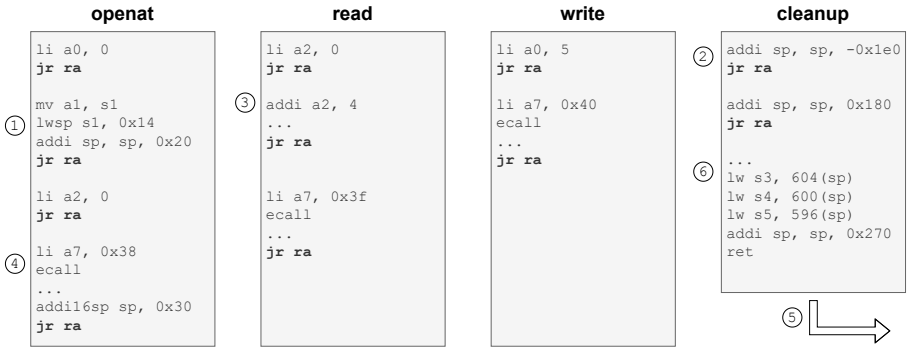

Fig. 9. JOP chain.

Several details are of interest in this gadget chain:

- Some gadgets have side-effects e.g. modifying the stack pointer or loading registers from the stack. While the former is fixed by the cleanup part of the JOP chain, the latter needs to be considered while forging our payload. For instance, `s1` is loaded from the stack (see ① in Figure 9) but we have to ensure it remains superior than `s0`.
- The gadget that subtracts `sp` (cf. ②) is not present in the original code: it is a valid instruction starting at an unexpected offset, often called shifted offset of misaligned instruction.
- For read and write syscalls, we need `a2` to be big enough to process the whole file (≈3 kB). However, we only found a gadget that increments `a2` by 4 (cf. ③). We used this gadget 651 times to suit our needs.
- For each system call, we found gadgets in the libc that set `a7` to the right identifier, and perform the syscall e.g. the `openat` one (cf. ④).
- After the cleanup step, we need to return to original code by inserting some address belonging to the Mongoose application (cf. ⑤). However, this address will be called as a regular gadget, thus pushing a new shadow stack entry. In order to avoid shadow stack detection, we must return to some point in the code that "never returns", e.g. in an infinite loop.
- As part of the cleanup step (and after having fixed the stack pointer), we used the entire epilogue of the function from which we hijacked the execution-flow as a gadget (cf. ⑥). This allowed to restore saved registers before returning to original code outside of this function. This gadget is the only one we used that comes from Mongoose. The same goal could have been achieved with gadgets found in the libc, but using the epilogue of the function we hijacked to pop its stack frame is very convenient. Moreover, the same technique could easily be applied while attacking other binaries.

Once the JOP chain has been designed, it can be encoded as a dispatch table, which is a sequential table containing the address of each gadget. In case of a gadget repetition, its address is included as many times as needed.

Running the attack. The last step is to assemble everything we have seen so far to craft the body of our malicious HTTP POST request, that we will send to the Mongoose web server in order to exploit it. This subsection describes the fully-fledged attack, which is also illustrated in Figure 10.

To hijack the execution-flow, we used the format string vulnerability to overwrite the Global Offsets Table (GOT) entry of the fflush function — called right after the vulnerable printf — with the address of our initializer gadget. Doing so, our initializer gadget will be given control when the program will make the fflush call.

Although the stack frame could be under the attacker's control for several reasons, in our case we also used the write-anything-anywhere primitive offered by the format string vulnerability to set the values loaded from the stack by the initializer gadget, e.g. we set the stack address that will be loaded into `s1` to

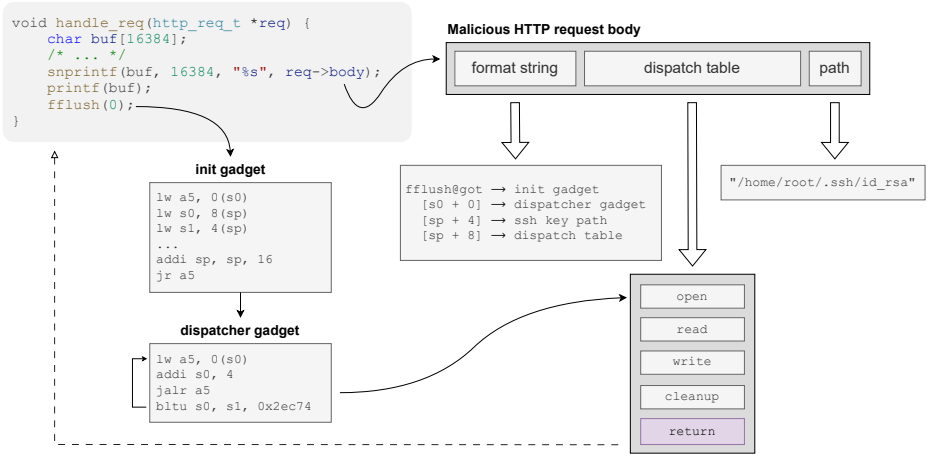

Fig. 10. Mongoose stealth JOP attack.

the path address in the request body. We then append the dispatch table and the path of the file we want to read next to the format string and perform the HTTP request.

While not related with JOP, in order for the attack to be stealth, we need to restore the fflush GOT entry we modified. To do so, a second HTTP request triggering the same vulnerability does the job. We can either patch the fflush GOT entry with the actual fflush address if we know it, or set it back to the default Procedure Linkage Table (PLT) stub address, to let the dynamic linker resolve its address again at the next fflush invocation.

4.1 Results and Limitations

By using the techniques presented in this section, we were able to steal the private SSH key of the root user stored on the target's disk. While not implemented in our target processor the attack should not trigger the shadow stack detection. The server still runs fine after the attack, and other clients can still interact with it normally.

As of today, it seems that landing pads could not be defeated with these techniques. To do so, one would need to use bigger gadgets — and eventually full functions — which may become impractical. However, landing pads come with a cost in terms of code size and execution time, that make them impractical for many embedded systems, hence our attack is mostly relevant for these systems.

4.2 Next Steps

As far as we know, there is no publicly available implementation of a standalone RISC-V shadow stack, without other CFI mitigations such as landing pads. For this reason, while we can theoretically bypass it, we were not able to test our

attack against an actual shadow stack implementation. This is left as future work.

5 Related Work

5.1 Building JOP Attacks

Brizendine et al. [2] proposed and implemented a method allowing building JOP gadget chains for the x86 architecture. The method relies on predefined gagdets of known characteristics, found in Microsoft Foundation Class (MFC). While this approach can reliably build JOP chains when known libraries are involved, it implies to update the tool catalog whenever these libraries are updated, and to perform in-depth analysis of them.

Other approaches try to build partial gadget chains by analyzing the whole used code, binary and libraries [22,16]. While some can integrate sub-chains of JOP gagdets within a ROP chain, none of them can build full JOP chains to our knowledge, making them easily detected by ROP-targeted countermeasures such as the shadow stack. Other tools able to help building JOP chains on RISC-V include ROPgadget[6] and radare2[7], which can search JOP gadgets but not build gadget chains on RISC-V architectures, as they have no method for discovering dispatcher gadget or initializer gadget. They are primarily designed to build ROP chains.

Gu et al. [12] identified a specific pattern of instructions allowing linking functional gadgets in RISC-V architectures, introducing the concept of "self-modifying gadget chain" to save and restore register values in memory. They also demonstrated the Turing-completeness of their solution. Adapting self-modifying gadget chain to JOP is indeed a promising solution to increase our capacity to build effective gadget chains. Jaloyan et al. [14] reached the same result by abusing compressed instructions (*overlapping*). Our attack also uses this approach, and applies it to JOP attacks.

Trampolines-based approaches are somewhat a missing link between ROP and JOP. A trampoline itself (an update-load-branch suite of instructions) is the ancestor of the dispatcher gadget and, instead of exploiting an arbitrary memory, uses hardware-maintained registers such as `ra` (return address register) to jump to the next functional gadget [5]. While they do not rely on return-specific instructions (which do not exist in RISC-V anyway), they do imply that large segments of the stack need to be corrupted, hence making them vulnerable to stack canaries and the shadow stack. Erdödi [8] proposed a solution to find classical dispatcher gadgets on x86 for different operating systems. As they are scarce, and trampolines patterns tend to be more common, the latter are still used [18]. In addition to providing a solution in RISC-V architecture for JOP gadget chaining, our discovery of the ADG greatly increases the number of available JOP gadgets, effectively making them as common as ROP gadgets and eliminating the need for trampolines.

[6] `https://github.com/JonathanSalwan/ROPgadget`.
[7] `https://github.com/radareorg/radare2`.

5.2 Defenses from CRA

Austin et al. [13] published the MORPHEUS II solution for RISC-V. This hardware-based solution aims at defeating memory probes trying to bypass address randomization by providing a reactive, fine-grain, continuous randomization of virtual addresses, as well as encryption of pointers and caches. This solution, while having a low overhead in terms of energy consumption and area, is quite intrusive in the hardware and may require efforts for certification in critical applications. While authors make no claim about stopping JOP attacks, probe-resistant ASLR may be difficult to bypass for an attacker.

Palmiero et al. [17] proposed a hardware-based adaptation of Dynamic Information Flow Tracking (DIFT) for RISC-V, with the ability to detect most function pointers overwriting, whether directly or indirectly, and in any memory segment, thus allowing blocking the attack at its initialization stage. Although this approach seems indeed powerful, it implies modification of RISC-V instructions behavior in I and M extensions for RISC-V 32 bits, as well as in the memory layout (by adding a bit every 8 bits of memory). Such modifications drift away from the RISC-V ABI.

De et al. [7] implemented a chip compliant to RISC-V, including a Rocket Custom Coprocessor (RoCC) which extends the RISC-V ISA with new instructions allowing safe operation on the heap. The authors ensure heap size integrity and prevent use-after-free attacks, at the cost of an increase of 50% of average execution time on their benchmarks.

6 Conclusion

Anticipating security vulnerabilities for RISC-V systems in order to identify and prevent possible attacks is an important challenge. Building attacks is a necessary step to test platforms and evaluate their attack surface, as adversary actors (*black hat* hackers) will eventually attack them. In this article, we contribute by demonstrating the feasibility and a practical way to realize jump-oriented programming (JOP) attacks, allowing for more extensive security testing.

We have introduced a new variant of dispatcher gadget, the *autonomous dispatcher gadget* (ADG), which greatly improves the RISC-V JOP attack surface by enabling the use of ROP gadgets with a JOP mechanism. While its rigorous validation against a CVA6 implementing a shadow stack is left as future work, we are convinced that it will be able to bypass shadow stack mitigation.

We have demonstrated a JOP attack on a RISC-V platform using a real world application commonly used in critical embedded systems: the Mongoose web server. After adding a single memory vulnerability, we were able to take control of the application in order to perform an adversary action, sending a private key to a remote attacker. Thanks to the large number of functional gadgets available in the libc through the use of the ADG, we were able to make the attack stealthy by restoring the nominal behavior of the application after the attack completition.

Next steps for identifying potentially practical JOP attacks include assistance in gadget finding and even automated chain building. There is a very impressive body of research on ROP chain building [22], that would be a good basis to build up automated testing frameworks for RISC-V application vulnerabilities to JOP. Likewise, studies like the one presented in this article will enable the development of better and more efficient countermeasures for the RISC-V architecture against JOP attacks and enhance control-flow integrity in general.

Acknowledgements. This work is partly supported by the French research agency (ANR) under the grant ANR-21-CE-39-0017. We thank Franck Viguier for his contribution to a preliminary version of this work.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: the 12th ACM Conference on Computer and Communications Security (CCS'05). p. 340–353. ACM (2005). https://doi.org/10.1145/1102120.1102165
2. Brizendine, B., Babcock, A.: Pre-built JOP chains with the JOP ROCKET: Bypassing DEP without ROP. In: Black Hat Asia (May 2021)
3. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Comput. Surv. **50**(1) (2017). https://doi.org/10.1145/3054924
4. Carlini, N., Wagner, D.: ROP is still dangerous: Breaking modern defenses. In: the 23rd USENIX Conference on Security Symposium (SEC'14). p. 385–399. USENIX Association (2014)
5. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: the 17th ACM Conference on Computer and Communications Security (CCS'10). pp. 559–572. ACM (2010). https://doi.org/10.1145/1866307.1866370
6. Cowan, C.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: the 7th USENIX Security Symposium. USENIX Association (1998)
7. De, A., Ghosh, S.: HeapSafe: Securing unprotected heaps in RISC-V. In: the 35th International Conference on VLSI Design and the 21st International Conference on Embedded Systems (VLSID'22). pp. 120–125. IEEE (2022). https://doi.org/10.1109/VLSID2022.2022.00034, https://doi.org/10.1109/VLSID2022.2022.00034
8. Erdödi, L.: Finding dispatcher gadgets for jump oriented programming code reuse attacks. In: the 8th International Symposium on Applied Computational Intelligence and Informatics (SACI'13). pp. 321–325. IEEE (2013). https://doi.org/10.1109/SACI.2013.6608990
9. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over ASLR: attacking branch predictors to bypass ASLR. In: the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16). pp. 40:1–40:13. IEEE (2016). https://doi.org/10.1109/MICRO.2016.7783743
10. Gilles, O., Viguier, F., Kosmatov, N., Gracia Pérez, D.: Control-flow integrity at RISC: attacking RISC-V by jump-oriented programming. CoRR (2022). https://doi.org/10.48550/arXiv.2211.16212

11. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: Practical cache attacks on the MMU. In: the 24th Annual Network and Distributed System Security Symposium (NDSS'17). The Internet Society (2017)
12. Gu, G., Shacham, H.: Return-oriented programming in RISC-V. CoRR (2020), https://arxiv.org/abs/2007.14995
13. Harris, A., Verma, T., Wei, S., Biernacki, L., Kisil, A., Aga, M.T., Bertacco, V., Kasikci, B., Tiwari, M., Austin, T.M.: Morpheus II: a RISC-V security extension for protecting vulnerable software and hardware. In: the IEEE International Symposium on Hardware Oriented Security and Trust (HOST'21). pp. 226–238. IEEE (2021). https://doi.org/10.1109/HOST49136.2021.9702275
14. Jaloyan, G.A., Markantonakis, K., Akram, R.N., Robin, D., Mayes, K., Naccache, D.: Return-oriented programming on RISC-V. In: the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS'20). p. 471–480. ACM (2020). https://doi.org/10.1145/3320269.3384738
15. Lu, T.: A survey on RISC-V security: Hardware and architecture. CoRR (2021), https://arxiv.org/abs/2107.04175
16. Nurmukhametov, A., Vishnyakov, A., Logunova, V., Kurmangaleev, S.F.: MAJORCA: Multi-architecture JOP and ROP chain assembler. In: the 2021 Ivannikov Ispras Open Conference (ISPRAS'21). pp. 37–46 (2021). https://doi.org/10.1109/ISPRAS53967.2021.00011
17. Palmiero, C., Di Guglielmo, G., Lavagno, L., Carloni, L.P.: Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for iot applications. In: the 2018 IEEE High Performance Extreme Computing Conference (HPEC'18). pp. 1–7. IEEE (2018). https://doi.org/10.1109/HPEC.2018.8547578
18. Sadeghi, A.A., Aminmansour, F., Shahriari, H.R.: Tazhi: A novel technique for hunting trampoline gadgets of jump oriented programming (a class of code reuse attacks). In: the 2014 11th International ISC Conference on Information Security and Cryptology. pp. 21–26 (2014). https://doi.org/10.1109/ISCISC.2014.6994016
19. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: the 2007 ACM Conference on Computer and Communications Security (CCS'07). pp. 552–561. ACM (2007). https://doi.org/10.1145/1315245.1315313
20. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: the 11th ACM Conference on Computer and Communications Security (CCS'04). p. 298–307. ACM (2004). https://doi.org/10.1145/1030083.1030124
21. Solar Designer: Getting around non-executable stack (and fix) (1997), https://seclists.org/bugtraq/1997/Aug/63
22. Vishnyakov, A., Nurmukhametov, A.: Survey of methods for automated code-reuse exploit generation. Programming and Computer Software **47**, 271–297 (2021). https://doi.org/10.1134/S0361768821040071
23. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++: A survey of vulnerabilities and countermeasures. Tech. rep., Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004), https://www.cs.kuleuven.be/publicaties/rapporten/cw/CW386.pdf
24. Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. IEEE Trans. Very Large Scale Integr. Syst. **27**(11), 2629–2640 (2019). https://doi.org/10.1109/TVLSI.2019.2926114