Ret-gadgets in RISC-V-based Binaries Resulting in Traps for Hijackers

Toyosi Oyinloye[1], Lee Speakman[2] and Thaddeus Eze[1]
[1]University of Chester, Chester, UK
[2]University of Salford, Manchester, UK
t.oyinloye@chester.ac.uk
l.speakman@salford.ac.uk
t.eze@chester.ac.uk

Abstract: The presence of instructions within executable programs is what makes the binaries executable. However, attackers leverage on the same to achieve some form of Control Flow Hijacking (CFH). Such code re-use attacks have also been found to lead to Denial of Service (DoS). An example of code re-use attack is Return Oriented Programming (ROP) which is caused by passing input crafted as chained sequences of instructions that are already existing as subroutines in the target program. The instructions are called gadgets and they would normally end with ret. The ret instructions enable the flow of hijacked execution from one set of instruction to another within the attacker's control. There could however be exceptions depending on the structure of the chained gadgets where the chained gadget fails to run its course due to inability of specific gadgets to replace the value in the return address (ra) register. The dangers of chained gadgets are not a new idea but the possibility for an attacker's gadget chain to fall into a trap during a ROP attack is not commonly addressed. In addition to this, recent studies have revealed that understanding the behaviours of gadgets would be useful for building information base in training machine learning (ML) models to combat ROP. This study explains the behaviour of certain ROP gadgets showing the possibility of occurrence of a loop in execution during exploitation. A sample program which accesses gadgets from the GNU C library (glibc) is used to demonstrate the findings. Gadgets identified with this possibility are poor for chaining as they do not contain instructions to load or move new values to the ra register and would produce unreliable exploits. This would result in a trap for the chained gadgets instead of arbitrary code execution, and DoS on the path of the user. This implies that the impact that a ROP chain could have on a targeted process does not only rely on the underlying system architecture but also on relies on the structure of the chained gadget. In this paper, the RISC-V architecture is the focus, new gadget finders (scripts are available) are presented, and sample of chained gadgets are analysed on a RISC-V - based binary.

Keywords: RISC-V; Return Oriented Programming; Denial of Service; Ret-gadgets; Code-reuse; Control Flow Hijack

1. Introduction

Ret-gadgets are the weapons of warfare in ROP (Shacham, 2007) attacks and as long as vulnerable programs remain, it is important to seek ways of providing protection for vulnerable programs. In the effort to protect vulnerable programs, it would be valuable to further examine the weapons of attack based on their underlying architectures and behaviour of the attack vectors. This would empower stakeholders to better understand what they are fighting against. Attackers of software have developed various techniques for CFH. While buffer overflow remains a fundamental route to complex attacks, further exploitations are made feasible due to executable instructions in the programs. Existing code content in executables offer functionality to the program but on the other hand, could become dangerous tools in the hands of hijackers. Existing code ending with ret referred to as gadgets are used by attackers to circumvent basic protection techniques like DEP/NX (Microsoft, 2018) and ASLR (Pax Team, 2003). The gadgets reside as subroutines in the text or shared library section of the original program. As the gadgets lie within, they can be accessed throughout the runtime of the program and are ready tools that attackers use for code-reuse attacks.

Code reuse attacks involve the use of existing code within executables to perform tasks that are different from the original intention of the programmer. Code reuse attacks could occur as CFH via buffer overflow which escalates into ret-to-libc and then ROP. The ret-to-libc could occur when attackers pass addresses of existing functions as input into a target while ROP could begin and end successfully with a chain of gadgets ending with ret, jump or a call as input via buffer overflow. A malicious input could be used in a buffer overflow followed by redirection of control flow to an injected malware or to an unintended destination within the target program. If the attacker passes an input that only contains addresses and values, they can achieve a ROP attack. The first address redirects execution to a different instruction and once that instruction is executed, it returns to fetch the next address to be executed until a full-blown ROP is achieved. A ROP chain would not always lead to a complete ROP attack but would still inhibit the purpose of the running process. This is because the completion of a ROP attack depends on the level of complexity of chaining the gadgets, as well as the behaviour of the gadgets that are included in the ROP chain. Availability of gadgets within a binary depends on the way the system architecture interprets the executable binary. The forms of the gadgets would then determine the possible variations in sequences of chained gadgets and ultimately the outcome of the ROP attack.

The possibility of complete ROP on different architectures has been demonstrated in various studies but the gadgets that are used in achieving the ROP are not widely studied particularly for the RISC-V architecture. For example, studies (Shacham, 2007; Checkoway et. al, 2010, Bletsch et al., 2011, Carlini & Wagner, 2014) on implementation of ROP attacks and possible means of mitigations for ROP (Abadi et al., 2005; Nui, 2014)have been presented with regards to programs built and run on the X86 and ARM CPUs. This is justifiable as X86 and ARM have been the commonly implemented systems for computer devices and smart devices respectively. In recent times, RISC-V, a new CPU architecture, has been developed and is fast gaining popularity particularly among producers of electronic voting machines, smart devices, personalised health management systems, etc. (IIT Madras, 2020), and recently laptop computers (RISC-V Community News, 2022). While the possibility of ROP across different architectures has been established, the complexity of ROP chaining, the variety of possible outcomes, and behaviour of ROP gadgets particularly on the RISC-V-based binaries are not yet widely explored. For this reason, this study focuses on the RISC-V architecture. Understanding the behaviour of various gadgets would be valuable for predicting the feasibility of a ROP attack on any architecture.

Gu and Shacham (2020), and Jaloyan et al. (2020) revealed that the underlying architecture of the executable program could impact the feasibility of ROP. The CPU interpretation of instructions could generate new instructions that could be useful as gadgets but not detectable by existing protections. In addition to this, the existing protections have not reasonably considered vulnerability of RISC-V-based binaries. Apart from the knowledge of underlying dangers of ROP on RISC-V, recent study by Koranek et al. (2022) also presents models that were developed with focus on ROP gadgets as valuable resource for ML towards detection of possible ROP. Understanding of the behaviour of gadgets would be useful in this regard.

A sample C program compiled in RISC-V was examined and two new gadget finders were built in the process of this study. The program is then exploited as the target for ROP attack with focus on gadget *getpid* which exists in glibc. The C program can access glibc during execution which indicates that the event presented here can be replicated in any other Linux on RISC-V-based binary. In addition to this, we highlight how the characteristics of ret instructions may vary as they tend to behave in line with the operations that they were originally built to perform. Secondly, we highlight that the addresses that are stored as return address require to be overwritten with the next destination in a ROP execution on the RISC-V-based binaries. The form of ret instruction that is encountered in the ROP chain would determine the next step that the ROP execution would take. This could be a success and progressive ROP attack or a failed attack which results in a trap for the hijacker, and a DoS for the user. With reference to these findings, we demonstrate entrapped ROP on the target. The outcome of this study is valuable information that could be harnessed for building ML technique towards detection of malware and CFH.

As software exploitation via ROP is an on-going challenge in software securities, existing mitigations for DoS caused from ROP is discussed. The remainder of this paper is structured as follows: Section 2 discusses related works. Section 3 presents details on the approach adopted in this study. Section 4 discusses the selected ret gadgets that were extracted from the sample program and our implementation of ROP using the gadgets. Section 5 discusses the findings. Section 6 is a conclusion and brief on future works.

2. Related works

There are previous studies which have discussed the use of ret gadgets for software exploitation in form ROP via CFH. One of the major studies around the concept of gadgets that resulted in turing complete ROP was presented by Shacham (2007) where ret-to-libc and Return Oriented Programming(ROP) were demonstrated. Shacham's study was demonstrated on the X86 architecture with the use of large combination of short sequences of instructions to achieve a ROP, while substantial tasks towards ret-to-libc were achieved using whole blocks of functions existing in libc library. Shacham's gadgets were extracted from X86 executable program through static analysis and using Galileo's algorithm. Similarly, other studies have presented variations of ROP on X86 in form of Jump Oriented Programming (Bletsch et al., 2011) and Call Oriented Programming (Carlini & Wagner, 2014)). While the gadgets used by Bletsch et al. consist of sequences of indirect jump instructions, gadgets used by Carlini and Wagner consist of instructions that end with indirect calls.

This study adopts methods from Shacham (2007), first by using static analysis via objdump tool and other Linux commands that are useful for static analysis, to fetch available gadgets but in addition, dynamic analysis via gdb was used to enable a view of the process during the implementation of the ROP exploits. We highlight the behaviour of specific gadgets which are available in glibc to demonstrate ROP based on chained sequences of instructions on a different CPU, which is RISC-V. Similarly to Bletsch et al., (2011) and Carlini and Wagner (2014), the gadgets in this study include jump oriented gadgets and gadgets ending with indirect calls.

Study by Checkoway et al.(2010) presents implementation of ROP using gadgets without returns on X86 and ARM architectures. According to Checkoway et al., there are two properties that make it possible for return instructions to be misused as gadgets in chained malicious input. First, the possibility of transferring control of execution via indirect jumps, and secondly, their ability to update processor state such that subsequent flow of execution is not transferred back to the same location. This study agrees with Checkoway et al.'s statement on these properties as a violation of the second property would effectively cause a loop in the execution which we consider in this study as a trap for the hijacker's malicious code. Apart from entrapping the ROP, it is also observed that this would lead to a DoS for the user. Checkoway et al. presented instruction sequences that behave similarly to returns. This was achieved by inserting gadgets that contained instructions that could act as trampoline for the ret without return instructions.

In this study, a similar method to that of Checkoway et al. (2010) is applied through the use of sequences of instructions called chargers or linker gadgets. The gadget that would then carry out the needed task of full exploitation is termed as functional gadget. In the implementation done in this study, the order of the sequence of instruction was a deliberate plan of inserting the *no-return* gadgets within the gadget chain to show how the loop is established. The terms chargers, linkers and functional gadgets were adopted from the theoretical plan for ROP on RISC-V by Deac (2022). Also, while Checokway et al., made use of the same algorithm as Shacham, the gadgets in this study were extracted using newly written scripts for use within RISC-V system.

Other similar works that studied exploitation of executable binaries via ROP are Gu and Shacham (2020), and Jaloyan et al. (2020) which both revealed that RISC-V is prone to a turing complete exploit via ROP. According to Gu and Shacham (2020), this could be achieved if an attacker uses gadgets retrieved from glibc in forming ROP chains. Jaloyan et al. (2020) also identified that the underlying structure of RISC-V could be an opportunity for attackers to retrieve gadgets that are not detectable by existing gadget finders or analysis tools. For this reason, there could exist hidden gadgets in the Control flow which would not be considered in the development of existing ROP mitigation techniques. Although this study does not emphasize on this fact, it is important to note that such gadgets continue to threaten the security of software.

A recent study by Koranek et al. (2022) also focused on ROP on RISC-V to proffer solutions towards detection of ROP on RISC-V by using deep learning (DL) models to distinguish features from execution trace. Koranek et al. analysed branch patterns in ROP as valuable information towards ROP detection. They presented models that were capable of predicting the classification of execution trace thereby enabling detection of possible ROP exploits. This study relates with Koranek et al.'s goals to contribute to malware detection and detection of possible ROP exploits. According to Koranek et al., DL and ML have been found to be useful for detecting malware particularly with the complexity of malware evolution that we are currently facing. The more information that is obtained and analysed from numerous samples of programs, the more empowered the ML training, testing and validation would be. The sample shown in this paper involves code from glibc which is normally accessible to any executable running on the Linux system. The form of exploit demonstrated can be replicated on other vulnerable executable programs compiled on RISC-V system. Results presented in this study would be valuable input for building ML development/training algorithms towards detection of ROP and other CFHs.

3. Approach to the study

In this study, technical steps were taken to acquire sequences of instructions, identify viable sequences of instructions, and build gadgets chains. Further steps include exploiting the target program with the chained ROP input to record and analyse specific outcomes of passing chained gadgets. We began by compiling the sample C program that were written purposely for the study of software exploitation. The programs are vulnerable to buffer overflow and ROP. We compiled the program statically such that the libraries are included within the executable program, the optimisation was set to 0. This is to give us more access to all the available sequences of gadgets, and lots of options to choose from. The program was compiled with gcc and analysed on the platform where it was compiled.

Once the program was compiled into executable, further steps were taken towards selecting/analysing viable gadgets using two new gadget finders, deliberately chaining gadgets in specific order, and passing the chained gadgets as input into the target to mount ROP. These steps were achieved based on possibility that selected gadgets would give an outcome that is different from the original purpose of the target. The selected gadgets were chained together tin the exploitation phase. *gdb* was used for dynamic analysis for observing execution and recording of the outcome of each exploit.

The behavioural pattern of each gadget chain was recorded highlighting specific order of chained gadgets that lead to traps for the ROP input, such that the execution continues to loop over the last bunch of instructions. It was identified that depending on the carefulness of the attacker with crafting the gadget chains, there could be three different possibilities. The input might cause the target to crash, or fall into a trap, or allow a complete execution of the instructions held in the gadgets addresses without the user knowing. All demonstrations in this paper were run on Linux RISC-V (RV64) architecture. RISC-V is not yet operational on computers. The study system is an emulated Linux Fedora RISC-V64 which was built on a QEMU emulator. The emulator resides on a Linux Fedora computer system.

3.1 Gadget finders

There were no RISC-V gadgets finders accessible during the course of the study, so we wrote two new scripts as gadget finders which made a variety of gadgets available. One of the gadget finders could find sequences of instructions ending with *ret* while the other could find sequences of instructions ending with *jal* or *jalr*. Instructions ending with ret is a general concept in execution but the *jal* and *jalr* are peculiar to RISC-V. *jal* and *jalr* both end up being interpreted as call to a function based on the return address. The gadget finders, *RETGadget* and *JALRGadgets* were written in Linux scripts specifically for RISC-V-based programs. They make use of static analysis tool *objdump* and other linux command line tools for extracting specific sequences of instructions. Extracted output get written into text files from which further analysis were done.

3.2 Extracting and Selecting gadgets

To extract gadgets from the sample program, the program was passed as argument to each of the gadget finders. Gadgets that could fit into the target frame were selected from the output files. Some of the extracted *ret* gadgets have attributes as identified by Checkoway et al. (2010) in the two conditions: (i) To transfer control flow via indirect jump, (ii) Overwrite previous return address to ensure that control flow is not transferred to the same location but some ret gadgets do not fulfil the second condition. Although they are not valuable for achieving a full ROP, they can be used to cause a DoS. These group of gadgets are the ones that result in the loop in the execution flow, whereby the rest of the gadget chain is trapped and unable to fulfil the purpose of the attacker. Aside from gadgets ending with *ret*, other variety of gadgets that were aimed for were those ending with *jal* and *jalr*. These gadgets are useful to give the attack chain more capacity and control They are useful for passing new values onto the stack to overwrite values initially held in general purpose registers.

3.3 Crafting the ROP chains

Once useful gadgets were selected, we mapped out ROP chains in various order based on the theoretical approach by Deac (2022). ROP on RISC-V is complex especially compared to ROP on x86. The stack on RISC-V is managed a bit differently and manoeuvring along the stack via a ROP chain requires lots of gadgets in the input byte stream. In addition to this, on RISC-V, *ra* (return address) register holds different values from time to time during execution. For this reason, lots of manipulations can be done to redirect the flow for as long as the *ra* register can constantly be overwritten. According to Deac (2022), the charger gadgets can be passed along with desired values in order to store new values and addresses in available registers, making it possible to achieve the ROP. The goal is to create a fake frame, particularly in form of a function epilogue. This tricks the process into assuming that the previous function has been completed and then returns to the stack to fetch the address of next instruction, in which case, execution jumps to another fake frame as crafted by the attacker. This would go on until the execution encounters an instruction that halts the execution.

The outcome that each of the gadget chains achieves depends on the intention of the attacker. For this study, some of the gadget chains were crafted such that the execution falls into a loop to show how the attacker's input is trapped, causing a DoS. Other options of chained gadgets are also demonstrated to show how the gadget chain could be executed to completion in two different categories. In all cases demonstrated in this study, two classes of gadgets classified by Deac (2022) were chained sequentially as functional gadgets, and charger gadgets. The functional gadgets will hold the instructions for the actual attack or purpose, while the chargers (linkers) gadgets will load the registers with addresses of the functional gadgets and other useful values. The linkers are the ones responsible for creating the fake frame that introduces a fake epilogue to the preceding function, while each of the functional gadgets would be called from the new fake frame. With the functional gadget, lots of other exploits or further creation of fake frames could be achieved. For this reason, viable ROP chains for RISC-V would normally be longer than that of x86.

3.4 Passing the input

The input is passed as byte streams stored in a file. As the sequence in which gadgets are crafted in RISC-V would determine the outcome of the exploit, specific sequences that enable the outcome that are desired in the context of this study are used. This has a lot to do with the value of the *ra* register that gets overwritten from time to time as execution steps into and out of library functions or other functions that get called within a function. The branching pattern can only be detected during dynamic analysis and remain undetectable to users as no feedback is written to standard output.

4. Ret Gadgets on RISC-V and implementation of ROP

In this section we demonstrate the compilation of the sample program, extraction of gadgets, selection of gadgets, and implementation of ROP using input crafted from combination of gadget sequences formed using the selected samples. The outcome for each exploit is also discussed. This results in three categories of ROP. The first category makes use of a gadget that enable the *ra* to be overwritten as well as ending with *ret*. The second category uses a gadget ending with *jalr* instruction which could be harnessed in performing some exploits or just manipulating registers. The third category are gadgets that do not overwrite the *ra* and thus lead to a loop in the execution. The ret gadgets works more efficiently within fake epilogues but the *jal* or *jalr* can be used more flexibly. For this reason, the linker gadgets were built out of ret instructions. However, gadgets with ret that don't have an instruction to load a fake *ra*, can lead to a trap or it could end in some unexpected behaviour of the executable. The *jal/jalr* instructions, unlike the *ret* would enable the jump and link of execution to address held in the specified register. The order in which the gadgets are arranged in the chain would also affect the outcome of the exploit.

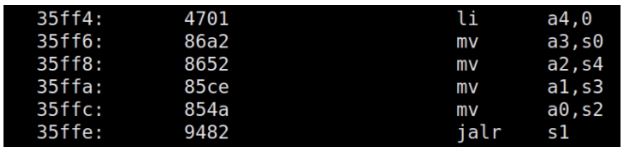
4.1 Compiling the programs

The program was compiled using gcc compiler and optimisation set to 0 to keep the code as large as can be and increase the possible available gadgets. The sample program is small but compiling it statically ensures that the libc code are also held in the resultant executable, making it larger than would normally be, and further increasing the number of available gadgets. The stack protection is bypassed in ROP, so the program was compiled with default setting including stack protections. The -g flag is also included to enable optimal dynamic analysis using gdb.

$ gcc stack_exploit_2.c –o stack_ex_s –g –O0 --static

4.2 Extracting the gadgets

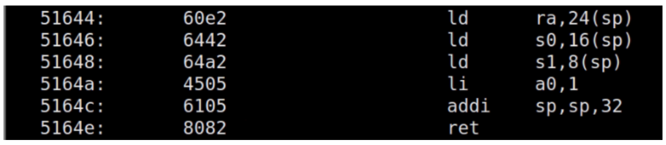
The compiled executable was passed as arguments while running each of the gadget finders. The new gadget finders extracted 26 *jalr* gadgets, 89 *jal* gadgets and 1,657 *ret* gadgets. Screenshots of the extracts for *jalr* and *ret* gadgets are shown in figures 1 and 2.



```
35ff4:      4701                li      a4,0
35ff6:      86a2                mv      a3,s0
35ff8:      8652                mv      a2,s4
35ffa:      85ce                mv      a1,s3
35ffc:      854a                mv      a0,s2
35ffe:      9482                jalr    s1
```

Figure 1: Extracted gadget ending with *jalr*

The gadget in figure 1 begins with instruction *li a4, 0* to load an immediate value of 0 into a general-purpose register a4 which would normally be used by the process to store a function argument. The next instruction *mv a3,s0* will copy the content of register *s0* into register *a3*. Registers in the range of *s0-s11* would normally hold values that persist after function calls. *s0* is the saved frame pointer and value in there should have been used to keep track of the stack. The next three lines of instruction are *move* instructions, copying the content of each of the saved registers *s4,s3*, and *s2* into *a2, a1* and *a0* respectively. The instructions are useful for ROP if exploit values have been passed earlier on into those *s0,s4,s3*, and *s2* registers. The last line of instruction is the jump and link register *jalr s1* which would write the value in *s1* into *ra*, and then pass execution to the instruction in there. Both *jal* and *jalr* instructions would pass new values to *ra*. *Jal* would pass a label and *jalr* would pass value stored in the stipulated register into destination register *rd* which is further interpreted as *ra*. This shows that *jal* and *jalr* gadgets are more useful as functional gadgets. As mentioned earlier, a charger gadget would be useful to control the registers before these types of gadgets can be applied as functional gadgets. The charger would preferably be a gadget ending with *ret*. An extract of gadget ending with *ret* is shown in figure 2.



```
51644:      60e2                    ld      ra,24(sp)
51646:      6442                    ld      s0,16(sp)
51648:      64a2                    ld      s1,8(sp)
5164a:      4505                    li      a0,1
5164c:      6105                    addi    sp,sp,32
5164e:      8082                    ret
```

Figure 2: Extracted gadgets ending with *ret*

The gadget in figure 2, ends with *ret* and begins with a *ld ra,24(sp)* load instruction. This is useful for taking control of the stack particularly the *ra* register which ultimately determines the branch pattern of the process. In other words, it can be triggered as a fake frame to trick the process into executing like a normal function. With the first load instruction, we can overwrite the value in *ra* with the address of the functional gadget that is shown in figure 1. The mapping on the stack for the value to be written to *ra* reserves a position of 24 bytes from the top of the stack for this. So that when we pass our input into the process, we must ensure that the address of the functional gadget is positioned correctly in the chain to get it stored in that location. Other values can be passed along with the new *ra*, and strategically linked to the chain to ensure that they are written into the current location on the stack which will enable the writing of new values to the registers *s0, s1, and a0*. The fourth instruction is *li* (load immediate) which loads 1 into *a0*. The line of instruction *addi sp,sp,32* deallocates the stack before the return is triggered.

4.3 Chaining the gadgets for specific goals

Several forms of gadgets were extracted. The type of gadgets available in a target program depends on the original purpose of the process itself and the CPU's way of interpreting the instructions. It is noteworthy to mention that library code is accessible to all the executables and gadgets can be fetched from there as well. Each category of gadgets that we have observed shows possible variations in the behaviour of gadgets. The *ret* gadgets, when they are used within fake frames, would have to return to address that is held in *ra*. While this is effective for establishing branching pattern, it could limit the instruction and restrict the flexibility in the manoeuvring for the attack. On the other hand, gadgets ending with *jal* or *jalr* can be used more flexibly. In demonstrating ROP here, we show how these instructions can be used with regards to their distinct advantages for changing values held on the stack and inside useful registers and then changing the behaviour of the process. Portions of the constructed input were also featured in recent work by Oyinloye et al. (2022). However, specific highlight here is on gadgets with *ret* that do not contain any instruction to load a fake *ra*, thereby leading to a trap for the input. Three examples of chained sequence of gadgets are demonstrated in the following subsections.

4.3.1 Chained sequence to create a fake frame

The ROP chain here is made of a charger and a functional gadget *exit* to show how target could be exploited with a safe exit at the end. This was passed as input that effectively hijacks execution and invokes an abrupt end. This also demonstrates the use of the linker beginning with an instruction to load a new address into *ra* register as the choice of gadget for the first element in the sequence. In figure 3, the charger creates the fake frame with values loaded or copied from one location or register to the other. Once execution is redirected, the new values are loaded into specified register *s1*, which was duplicated into *a0,* another register *s0*, and then new value for return address is written into *ra.* In this case, it points to the exit instruction.

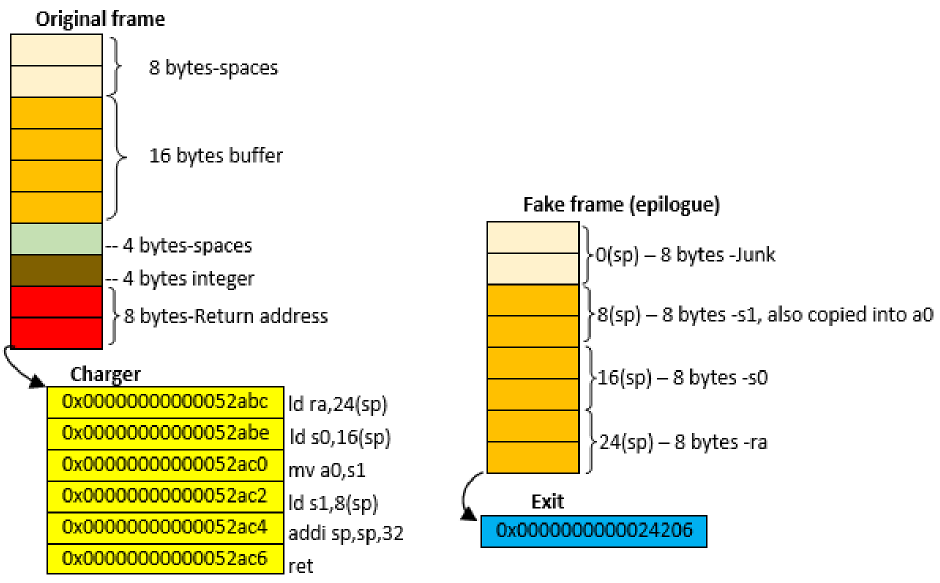


Figure 3: **Chaining gadgets to build fake frame and exit abruptly (Oyinloye et.al., 2022)**

This input in the first instance, overwrites the original *ra* with the address of the charger so that the fake frame will be running in a different address with different *sp* in the target process. Within the charger frame, a new *ra* is loaded with the address of *exit* using the ld *ra,24(sp)* instruction at the top of the charger. Other registers are loaded with junk using the charger and the input is passed into the stack of the target via the buffer. This results in an abrupt stop in execution.

4.3.2 Chained sequence to invoke a functional gadget

In this category, the ROP chain is extended by including a sequence of instructions ending with *jalr* and then an *exit*. For this example, we use a different charger. This is so that there could be more registers to manipulate by writing new values into them and then using a functional gadget to invoke the address that is written from the charger. This demonstrates how a functional gadget can be worked into the ROP to perform more exploit. The input passes values into *ra* & *s0-s2*. In the new frame, which is created by the charger gadget, *ra* will be the address of the functional gadget. *s0* and *s1* are filled with junk. The address of *exit* is then written to *s2*, to be called later from the functional gadget. The functional gadget duplicates the value in *s1* into *a0* and then *s0* into *s3* as shown in figure 4. This step is useful for the attacker to pass values into the registers that they might find useful for the exploit.

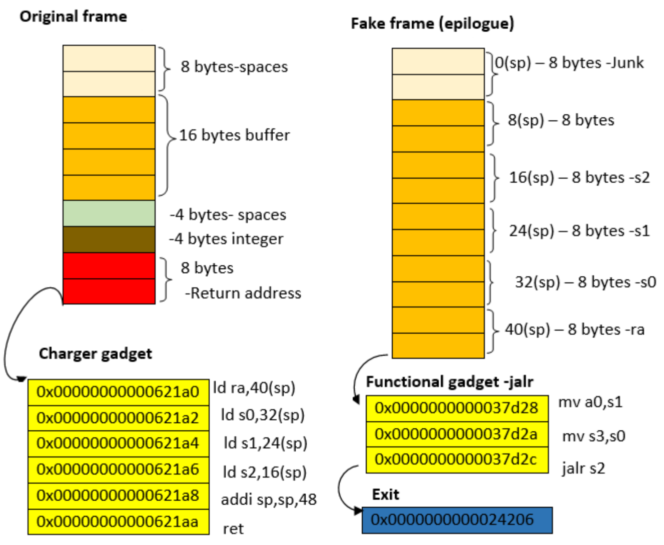
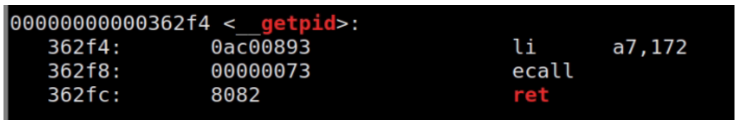


Figure 4: **Chaining gadgets to invoke a functional gadget**

4.3.3 Chained sequence to invoke a loop

ROP might not always end safely. The chained example in 4.3.2 can be transferred into a loop instead of *exit* if a gadget that does not include an instruction to overwrite the previous *ra* is inserted somewhere in the chain. When the functional gadget *getpid* (figure 5) was attached just after the *jalr* gadget, the program crashes because the *ra* returns back into the next instruction after *0x0000000000037d2c*. On the other hand, when the *jalr* gadget was taken out of the chain and the *getpid* gadget was chained directly to the charger, the return instruction directs execution back to the top of the *getpid* gadget and continues in a loop as shown in figure 6. This was observed in gdb but when the ROP chain was passed into the target at the command line, the execution only appeared to the user as if it was hanging. The execution did not crash but the process was not ending normally, resulting in a DoS.



```
00000000000362f4 <__getpid>:
   362f4:       0ac00893                li      a7,172
   362f8:       00000073                ecall
   362fc:       8082                    ret
```

Figure 5: **Extracted gadget *getpid***

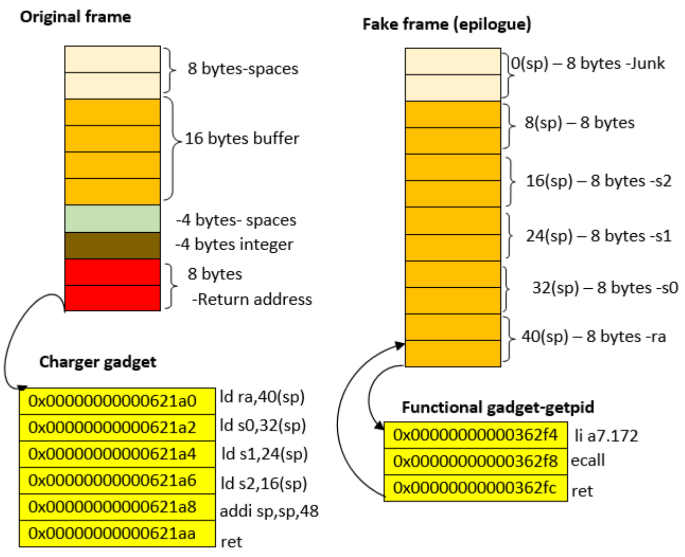


Figure 6: **Chaining gadgets to invoke a loop**

5. Discussion

The three categories of gadgets examined have shown different possible outcomes of ROP attacks based on the types of gadgets, the behaviours of each of the gadget, and the order in which the gadgets are chained. It could result in a full-blown exploit, a crash, or a loop. As for the *ret* gadgets, the expected operations are to pull address from the *ra* and jump into the address that was pulled to continue execution. This is what usually entails for return instructions but the peculiarity in RISC-V is that *ra* is a caller saved register. This means that upon each call to a subroutine, the return address is stored in the *ra* register. For this reason, the ROP chain needs to be crafted such that the intended return address for each subroutine is specified as demonstrated in figures 3 and 4. From the example in figure 6, it is highlighted that if the gadget does not include instructions to overwrite the value of *ra* then the execution would return to the value which has remained in the *ra* forming a loop in the execution. The same outcome was observed for *getgid* samples and could be for any gadget that does not enable the ROP chain to supply a new *ra*. The *getpid* and *getgid* gadgets exist in glibc and if used in this manner could become dangerous for the target.

None of the outcomes observed in section 4 is desirable on the path of the user. A loop in execution with no feedback to the user, is dangerous and could linger before it is identified. For the attacker, if they intended to achieve further exploits, passing execution to such gadgets would become an entrapment that truncates further execution of the ROP chain. The main finding here is that gadgets have peculiar behaviour based on the type of instruction at the end, the possibility of overwriting the *ra*, and the order in which the gadgets are chained. This could be used in predicting the outcome of a ROP attack. Since the sample gadget used to cause a loop in execution here was picked from glibc, it is evident that this understanding could cut across other programs in the Linux environment and can be used broadly as information for training ML models in recognising possible ROP attacks.

Exploitation via ROP is still a challenge in software securities. Possible mitigation for the DoS demonstrated here is demonstrated by Oyinloye et al., (2022) using a watchdog monitoring process. The watchdog monitoring is based on timing-out depending on the log taken from the previous function epilogue. According to Oyinloye et al., the watchdog monitoring system can identify a looping ROP and stop or prevent DoS. A full exploit which would normally end undetected can also be stopped particularly if it extends reasonably for the watchdog to trace the time lapse after the last legitimate function. From the study, it is observed that ROP chains in RISC-V would require to be longer for any reasonable exploit to be achieved. It is considered that such long chain might be traceable by the watchdog monitoring process before a full exploit is achieved.

6. Conclusion and future works

ROP on RISC-V is not straightforward. The behaviour of gadgets in a RISC-V based process would result in different outcomes. While there are existing mitigations, it is important to consider the behaviour of gadget chains particularly when they access glibc. More vulnerable programs can be evaluated in order to obtain valuable information which could be harnessed in building algorithms for ML towards intrusion detection. More samples of exploitation might also reveal further interesting details. Future works would compare gadgets extracted from various programs that are known to be vulnerable to ROP including remotely executed programs. These programs can be compiled on RISC-V and X86, and then exploited with various sequences of ROP chains to observe the outcomes. A comparison of the behaviours of similar gadgets on these two platforms would be useful in providing a cross platform information for ML/DL models or other Control Flow Integrity concepts for combatting ROP. Extracts from the new gadgets will also be analysed to identify the reliability of the gadgets for ROP attacks. This would involve considerations under different compilers as well as different system architectures.

References

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z., 2011. Jump-oriented programming: a new class of code-reuse attack. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security,* March.pp. 30-40.

Carlini, N. and Wagner, D., 2014. ROP is still dangerous: breaking modern defenses. *SEC'14: Proceedings of the 23rd USENIX conference on Security Symposium,* August.p. 385–399.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A-R., Shacham, H., and Winandy, M., 2010. Return-oriented programming without returns. *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security,* October.pp. 559-572.

Deac, B., 2022. *InfoSec Write-ups.* [Online] Available at: https://infosecwriteups.com/return-oriented-programming-on-risc-v-part-1-dd9817b52d2b[Accessed 25 06 2022].

Gu, G. and Shacham H., 2020. No RISC No Reward:Return-Oriented Programming on RISC-V. 29 July.

IIT Madras, 2020. *IIT Madras, Indian Institute of Technology Madras.* [Online] Available at: https://www.iitm.ac.in/happenings/press-releases-and-coverages/iit-madras-develops-and-boots-moushik-microprocessor-iot[Accessed 08 July 2022].

Jaloyan, G-A., Markantonakis, K., Akram, R. N., Robin, D., Mayes, K., and Naccache, D., 2020. Return-Oriented Programming on RISC-V. *ASIA CCS '20: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security,* October.p. 471–480.

Koranek, D.F., Graham, S. R., Borghetti, B.J., and Henry, W. C., 2022. Identification of Return-Oriented Programming Attacks Using RISC-V Instruction Trace Data. *IEEE Access,* Volume 10, pp. 45347-45364.

Microsoft Coporation, 2018. *Microsoft Documentation.* [Online] Available at: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention[Accessed 22 April 2020].

Oyinloye, T., Speakman, L., Eze, T., and O'Mahony, L., 2022. Watchdog Monitoring for Detecting and Handling of Control Flow Hijack on RISC-V-based Binaries. *International Journal of Advanced Computer Science and Applications(IJACSA),* 13(8).

Pax Team, 2003. *ASLR documentation.* [Online] Available at: https://pax.grsecurity.net/docs/aslr.txt[Accessed 22 April 2020].

RISC-V Community News, 2022. *RISC-V.org/blog.* [Online] Available at:https://riscv.org/blog/2022/07/deepcomputing-and-xcalibyte-open-pre-orders-for-first-native-risc-v-development-laptop-quantities-limited-xcalibyte-and-deep-computing/[Accessed 31 August 2022].

Shacham, H., 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *ACM conference on Computer and communications security,* pp. 552-561.