CV32A6 Softcore Hackathon

Téo Biton
Nantes Université
Nantes, France
teo.biton@etu.univ-nantes.fr

Louison Gouy
Nantes Université
Nantes, France
louison.gouy@etu.univ-nantes.fr

***Abstract*—This paper presents the results obtained during the third RISC-V national student contest organized by Thales, the GDR SOC² and the CNFM. The goal of the contest is to defeat buffer overflow based attacks from the Runtime Intrusion Prevention Evaluator (RIPE) testbed on a CV32A6 core geared with a ZephyrRTOS. There are ten attacks picked from RIPE with different parameters.**

After introducing the test environment, the ten attacks will be presented as well as the proposed countermeasures. The different modifications made to either the Real-Time Operating System (RTOS) or to the CV32A6 core will be demonstrated. To ensure that the system is still viable, performance test results and hardware resources statistics are also explicited.

***Index Terms*—RISC-V, security, embedded systems**

I. Introduction

In 2023, Master 2 students were met with the opportunity to participate in a RISC-V national student contest. It was organized by a French tech company, Thales, as well as two reseach groups: GDR SOC² and the CNFM. The purpose of this contest is to challenge students with software attacks executed on a RISCV-V Central Processing Unit (CPU), exploiting a buffer overflow vulnerability. They must come up with solutions, in software or hardware, to counter these attacks. The platform of the contest consists in executing the attacks on a CV32A6 CPU emulated on a Field-programmable gate array (FPGA) board. Moreover, to have more potential countermeasures available, it is equipped with a ZephyrRTOS Operating System (OS).

This paper proposes to present our process during this contest:

- Overview of the RIPE testbed and of the attacks
- Our approach during the contest and the obtained countermeasures
- The results obtained and our analysis

II. The attacks testbed

We had to counter 10 attacks from the RIPE testbed. These ten attacks exploit a buffer overflow in different ways, because they use different parameters. Before introducing our countermeasures, let us present our understanding of this vulnerability and of the testbed.

Thales, GDR SOC², CNFM

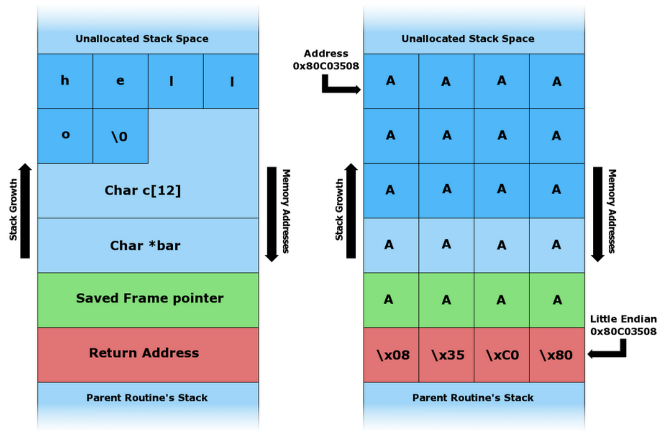

Fig. 1: Stack buffer overflow [1]

A. Buffer overflow vulnerability

A buffer overflow is a type of software vulnerability that occurs when a program tries to store more data in a buffer than it can hold. This can happen when a program is not properly checking the size of the input it receives, and as a result, the excess data overwrites adjacent memory locations. It can cause the program to crash or even be exploited by an attacker to execute arbitrary code. The most famous examples are stack-based buffer The stack is a region of memory that stores temporary data, such as sequences of function calls and local variables. When a buffer overflow occurs on the stack, it can overwrite adjacent data or, in the worst case, the return address. In the case of an attack, it is possible to write a well-chosen address to redirect the program flow, in order to execute injected code.

The figure 1 illustrates on the left a normal usage of stack buffer. On the right, it shows how a venerable buffer can be used to overwrite the return pointer. A twelve byte buffer stored in the stack is accessed without a boundary check. Writing beyond its capacity allows the return pointer value to be replaced by the buffer address. At the end of the function, the program will want to return to the caller but will jump to the buffer instead. The 'A' character represents instructions inserted by the attacker. The program flow is redirected to an injection code.

[1]Figure 1 from Own work (Original text: self-made) by Michael Lynn, 5 April 2007

However, buffer overflows were shown to also occur on the heap. The heap is a region of memory that is used to dynamically allocate and deallocate memory at runtime. Heap-based buffer overflows occur when a program writes more data to a buffer on the heap than the buffer can hold. Unlike stack-based buffer overflows, heap-based buffer overflows do not usually cause the program to crash, but they can be hijacked to execute arbitrary code. This is because the heap does not store a return pointer but can still be used to overwrite internal structures such as linked list pointers.

B. RIPE

The use of buffer overflow based attacks has been clearly documented for many years. Despite this, the subject of countermeasures remains active. In order to evaluate their performance, Wilander and Kamkar published in 2002 an indicator with 20 types of attacks to thwart [4]. Thus, a score is given to the solutions according to the number of neutralized attacks. Over the years some of them have reached a coverage rate of 100%. But on the other hand, the exploitation of buffer overflows has grown. It was therefore necessary to update this evaluator with the new findings. This is the work presented by John Wilander and Nick Nikiforakis in 2011 titled RIPE: Runtime Intrusion Prevention Evaluator [3]. It is a strong reference of this article and particularly of this section. The idea of this testbed is essentially to propose a more ambitious scale by going from 20 to 850 forms of attack. The article then submits some solutions to their new evaluator to expose their vulnerability. A brief presentation of each runtime buffer overflow prevention which makes it a partially accessible article for newcomers.

A RIPE attack definition is based on five different dimensions such as the buffer's location in memory, target code pointer, overflow technique, attack code and function abused. Instead of going through all the dimensions like does the RIPE paper, this section will present them based on the ten attacks of the contest.

1) Bypassing rights with buffer overflows: The first attack is the most classic form of buffer overflow. A table is defined in a function and the size is not control before calling the venerable function `memcpy`. By writing the right value while overwriting the return address, it is possible to redirect the execution flow to point in the table itself, where we had injected a shell code. It is particularly simple to implement. The main difficulty is to overwrite precisely on the return pointer, which can be corrected by adding a set of NOP instruction. There are however two main flaws in the attack. The first is that it overwrites everything between the beginning of the vulnerable array and the return pointer. And the second is that it executes code on the stack, a memory space that is not intended for this purpose. It is on these two defects that the associated countermeasures are based.

The second attack is very similar to the first. It is still a case of overwriting a branching address from a venerable buffer. The difference is the code pointer which is now a user defined structure. The code below extracted from RIPE sources is the declaration of the vulnerability.

```c
/* Technique : Function Pointer */
int (* stack_func_ptr)(const char *);
/* Location overflowed Buffer */
char stack_buffer[1024];
```

The weakness shown here is a buffer next to a function pointer. By overflowing the array, the function pointer will we overwritten. Next time the program jumps to its address, the flow is redirected to the programmer shell code injected in the buffer. In the previous attack, the return pointer was completely hidden for the developer. It was created at compilation time and its location was perfectly known to the system. This time, the vulnerable structure can be anywhere in the program. It makes the overflow more difficult to catch for a potential protection method. However, it has the same flaw, a memory space where the structure is stored is not intended to be executable.

This third attack was probably born to bypass the countermeasures of the first attack capable of detecting overflows on the return pointer. Indeed we had specified that its default was to overwrite everything between the beginning of the vulnerable buffer and the return pointer. This time the method is a bit more complicated but allows you to target precisely the area you want to overwrite and easily target precisely the return pointer.

```c
/* Example vulnerable program */
int f (char ** argv){
    /* Useless variable */
    int pipa;
    char *p, a[30];

    p=a;

    /* vulnerable strcpy */
    strcpy(p,argv[1]);
    strncpy(p,argv[2],16);
}
```

This example of a vunerable program above is quoted from [6]. In an environment using no protection mechanism the vulnerable *strcpy* would have allowed the return pointer to be rewritten. With elementary protection, like a stack canary, the attack would be immediately spotted and controlled. To bypass this, the 'p' pointer is first rewritten with the address of the return pointer. During the *strcpy* (safe), the memory area is no longer the character array but the return pointer. Thus the rewriting is perfectly controlled and the value of the canairy remains unchanged. The shell code injection can be done in the first *strcpy* and stored in 'a'. This is again the flaw in this attack as it involves injecting code into the stack. It should be noted, however, that the conditions required to execute this attack are more complex than the previous ones.

The fourth attack is performed in an other memory segment than the stack. It consists in the leakage of a variable stored in the heap, by executing a function in the heap. The *sprintf* function copies a variable to another by printing directly into it, with no input on the variable sizes. This attack reaches

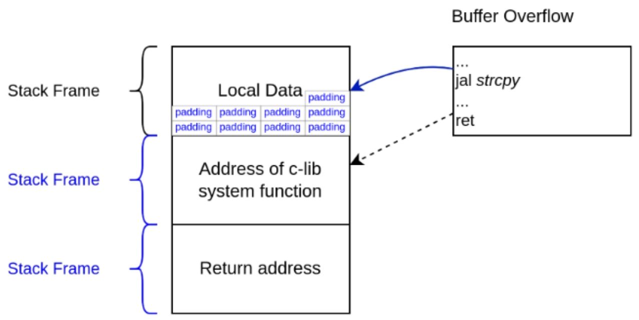

Fig. 2: Return2libc example in the stack

addresses beyond the bounds of the buffer, resulting in access to otherwise unreachable data. Unlike previous attacks, which were of privilege escalation type, this attack leaks data from memory regions that should not be accessed.

2) Code-reuse attacks: The remaining attacks are of a different kind: they are known as Code-Reuse Attack (CRA). Three of them use *return2libc* as injection parameter. The remaining uses Return Oriented Programming (ROP). A CRA is a software exploit in which an attacker takes control of the execution flow through existing code with a malicious result. ROP and *return2libc* are types of CRA, but many other exists like Jump Oriented Programming (JOP), or Call Oriented Programming (COP), which will not be developed here. A CRA is not in itself a primary attack, but relies on an earlier memory corruption of the stack or the heap. These type of attacks is totally different than data injection attacks like the first ones; it bypasses most countermeasures like Data Execution Prevention (DEP) or $W \wedge X$ pages.

The simplest CRA are *return2libc*, where the execution flow is redirected towards standard C libraries through a single function call. The redirection is made by overwriting the return address or a function pointer contained in the stack. For example, if the attacker knows its address, it can add a stack frames that call the standard C function *system()* that can execute commands. An example of such an attack is provided in figure 2.

More sophisticated CRA following the same principle, the most common being ROP. It consists in using gadget chains: gadgets are small snippets of code found within the application that end with a return instruction. ROP mechanism consists in linking these gadgets, all ending with a linking instruction which will pop the corrupted stack and then redirect the flow to the next gadget. The attacker can then run an arbitrary sequence of legit instructions, these gadgets can run a malicious, potentially a Turing-complete application within the application. Figure 3 is an example of a ROP attack, performing a read in memory. Such an example is shown in details in [7].

III. Approach

To counter these different attacks, there exists multiple solution, in software or in hardware. In this section, we

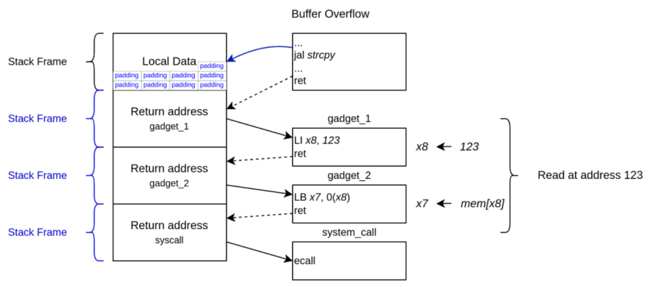

Fig. 3: Execution flow of a ROP attack

will show the three main countermeasures that we explored during the contest: one purely software, one both software and hardware and one purely hardware.

A. Stack canaries

The basic stack overflow attack overwriting the function pointer was presented above. The drawback coming with it was also mentioned. It overwrites everything between the beginning of the vulnerable buffer and the return pointer. To prevent this, a countermeasure was introduced named *canary* for the analogy to a canary in a coal mine[2]. The idea is to insert a randomly generated value just above the return address in the stack. Before branching, the value will be compared with a copy stored in a safe memory region. If they are identical, the branch instruction will be executed. If they are not, the return pointer has been overwritten, a specific behaviour can be chosen such as aborting normal execution. Stack canaries are inserted during compilation using the `-fstack-protector-all` flag with gcc. As an example, we can compare two compilation results of the same C program but with and without the stack protector. The C code is as simple as possible, a function that returns a static value. It was compiled using *riscv64-unknown-elf-gcc*, RISC-V's most commonly used compiler, to produce RISC-V assembly.

```
int main (void){
    return func();
}
```

This simple program is compiled with the following command.

```
riscv64-unknown-elf-gcc -O2 -S -fstack-protector-all canary.c
```

The result is shown below, left without the stack protector and right with it.

As we can see, the program is simple, the compilation only results in storing 0 in a register and returning. The stack protection adds more complexity. The first five instructions form the stack and store the value of the canary at Stack Pointer (SP)+8. This value is controlled by the conditional

[2]Domestic canaries were used in coal mine as carbon monoxide detectors. This expression in now used to describe someone/something that warns people of danger.

```
main:
li a0,0
ret
```

(a) Without stack canaries

```
main:
lui a5,%hi(__stack_chk_guard)
ld a4,%lo(__stack_chk_guard)(a5)
addi sp,sp,-32
sd ra,24(sp)
sd a4,8(sp)
ld a4,8(sp)
ld a5,%lo(__stack_chk_guard)(a5)
bne a4,a5,.L5
ld ra,24(sp)
li a0,0
addi sp,sp,32
jr ra
.L5:
call __stack_chk_fail
```

(b) Stack canaries enabled

branch instruction `bne`. If the comparison is not successful, execution does not continue but jumps to a function `__stack_chk_fail` instead. Otherwise, the program continues. The instruction overhead is clearly visible in this simple example. It is difficult to give a precise percentage of the canary overhead because it depends strongly on the program and especially on the number of function calls for example.

B. Data execution prevention

In order to increase security and reliability in program execution, the RISC-V Instruction Set Architecture (ISA) specifies an optional Physical Memory Protection (PMP). This hardware unit allows to define legal type of access to a precise memory region among read, write and execute. Access right can also differ from privilege modes. [2] One of the key parameters of the PMP is the granularity, the size of the minimal block to which a property can be assigned. In the RISC-V specification up to 64 PMP entries are supported. The CVA6 implements 16 of them based on the register map. To minimize the context switch time the configuration registers are packed in groups of four.

The definition of each configuration 8 bits register is given below. A second Control and Status Register (CSR) is needed to specify the memory location we want to protect. It contains bits 33–2 of a 34-bit physical address.

The way the region is defined from this address is settable in the `pmpXcfg` register. The proposal of the PMP is to assert allowed operations among read, write and execute, respectively : R, W and X. When one of these field is cleared, the corresponding access type is denied. For example, to configure a non-executable data region, the right combination is R=1, W=1 and X=0. Each PMP unit can be configured independently of others. If forbidden access occurs an access-fault exception is risen. The field L in this register gives us the ability to lock the PMP configuration for security and

7	6 5	4 3	2	1	0
L	0	A	X	W	R
1	2	2	1	1	1

Fig. 4: PMP configuration register (pmpXcfg) format.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, $\geq$8 bytes

TABLE I: Encoding of A field in PMP configuration registers.

reliability reason. If this bit is set, only a processor restart can allow a new write instruction in `pmpXcfg`. The A field in a PMP configuration register entry defines the address-matching mode used for the corresponding PMP address register. It is encoded on two bits so there are four different values for the A field as summarized in table I. OFF means that the PMP entry is not being used. TOR means top of range and the configuration is set for every address between $pmpaddr_n$ and $pmpaddr_{N-1}$: it uses two pmpaddr registers. NA4 is not being considered here sice we have a PMP granularity superior to 2. Hence, the last option is Naturally Aligned Power of Two (NAPOT). This is specifically used when the region' size is a power of two. Using the base address specified in a pmpaddr register, the $log_2(size) + 2$ first bits of the register will be masked with ones and preceded by a zero. This address matching mode uses less pmpaddr registers but needs more calculation time.

In the context of attack number 2 specified by a function pointer overflow on the stack, the idea with the PMP is to identify the memory space allocated to the stack of our application. With this, we hope to define it as non-executable and thus prevent code injection into the critical region. We know for a fact that this would stop the attack since PMP violations are trapped at the processor level.
Moreover, this could cover attack 3 et 4 that are data injection in the heap as well.

C. Return Address Protection

Many countermeasures exist to defeat this CRA. The "simplest" one is Address Space Layout Randomization (ASLR), consisting in randomizing the base address of memory segments so that the attacker can not directly find the gadgets. It makes the attack more difficult to craft. In the case of RIPE, the attacker is already within the application and already has access to the function pointers, making ASLR useless. Indeed, *RIPE is a process that attacks itself calculating the needed offsets within its source code* [3]. Hence, we did not develop that approach any further.

Another solution is named Control-Flow Integrity (CFI). It dictates the path followed by the software during execution, based on a Control Function Graph (CFG). The CFG of an application is determined ahead of time. During runtime, each function call and return instructions are checked to determine whether they are legit CFG edges or not. This approach, paired with a shadow stack, makes all kind of CRA

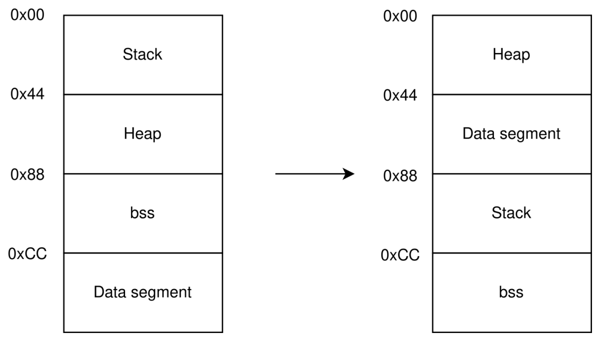

Fig. 5: Simplified example of an ASLR implementation

impossible to implement; however, it leads to a significant fall of performance.

That is why our first approach to defeat CRA is the use of a shadow stack alone. A shadow stack is a module implemented in the architecture, accessible only with a pop/push mechanism. It is used whenever a function call or a return instruction happens. We are interested in the property that functions do indeed return to the instruction following their call. A possible solution to verify such a property is to maintain a shadow stack. The processor pushes the expected return address onto this stack for each function call and pops it whenever it returns. If the address a function tries to return to, using the regular function stack and return address register, is not equal to the one on top of the shadow stack, we can deduce that something went wrong and react accordingly.

The CV32A6 is equipped with a return address stack, as dictated by the RISC-V specification [1]. In the RISC-V ISA, Jump And Link Register (JALR) instruction is used for function calls and returns. A jump destination is passed as source register `rs1` and the next instruction after the jump (pc+4) is stored in `rd`. The standard software calling convention uses `x1` as the return address register and `x5` as an alternate link register. Register `x1` is often mentioned as `ra` for "return address". Hence, when a JALR instructions is being executed by the CPU, the Return Address Stack (RAS) is updated:

rd is `x1`/`x5`	*rs1* is `x1`/`x5`	*rd*=*rs1*	RAS action
No	No	–	None
No	Yes	–	Pop
Yes	No	–	Push
Yes	Yes	No	Pop, then push
Yes	Yes	Yes	Push

TABLE II: Return-address stack prediction hints encoded in the register operands of a JALR instruction. [1]

This means that we have a secure backup of return addresses directly in the architecture. However, it is only used for branch prediction and it's not actually used to check the integrity of the return address. This is why we want to modify the RAS architecture by making it more secure.

The "secure" RAS, that we'll call a shadow stack, is able to detect when the current fetch address is different than the one stored in the stack, when a return instruction occurs.

This first base can then be extended to cover more than just the return pointer: with CFI or landing pads, both being currently defined for the RISC-V architecture in a specific task group.

IV. Results

We've managed to defeat attacks 1, 5 and 9 with security canaries. In fact, they share the same feature that is overwriting the return pointer. Even if attacks 5 and 9 are more complexe CRA, they redirect the execution flow using the direct technique. This necessarily implies the substitution of the canary value. Our results are presented in table III.

number	technique	injected by	code ptr	loc	fonct
🟩 1	direct	no nop	ret	stack	memcpy
2	direct	no nop	funcptrstackvar	stack	memcpy
3	indirect	no nop	funcptrstackvar	stack	memcpy
4	direct	data	var leak	heap	sprintf
🟩 5	direct	ret libC	ret	stack	memcpy
6	indirect	ret libC	funcptrheapvar	heap	memcpy
7	indirect	ret libC	structfuncptrheap	heap	homebrew
8	indirect	ret libC	longjumpbufheap	heap	memcpy
🟩 9	direct	rop	ret	stack	memcpy
10	direct	rop	structfuncptrheap	heap	memcpy

TABLE III: Contest results

Unfortunately, we have some good results that we can't complete. By enabling the PMP regions in the architecture, we were able to apply specific rights to defined memory locations in software. When executing instructions in these areas, an exception would occur. Applicative tests with Zephyr demonstrated our ability to implement this protection on known addresses. However, we could not identify at runtime where the user stack or the heap where instantiated. This could have covered attacks 1, 2 and 3; possibly attack 4 by applying read or write rights to certain regions.

We also worked on the hardware implementation of a shadow stack. Compliant to the RISC-V specifications, a RAS is present in the CV32A6 architecure; however, its data is not used for security purposes. By adding a simple comparison, we were able to detect when the return address was overwritten in bare-metal simulation. However, the RTOS layer added a certain complexity and all attempts to use this comparison failed due to false positives. This could have covered attacks 1, 5 and 9, like stack canaries, but with a hardware modification that would also lay a first ground for more sophisticated countermeasures.

V. Conclusion

In the context of the third RISC-V national student contest organized by Thales, the GDR SOC² and the CNFM, a study on BOF countermeasures was presented in this paper. After an introduction to how BOF works, it covers stack canaries, a solution for detecting return pointer overwriting. Although relatively easy to implement, stack canaries are not ideal because of the significant overhead they introduce, as we saw in a basic example. This is the reason why we considered implementing a shadow stack, a hardware countermeasure that protects the return pointer by making a copy and then a comparison with the regular stack. Convincing simulation results were obtained but the use of Zephyr systematically generated false positives and processor termination. Also, the PMP was introduced as a solution to prevent data execution. It is an internal feature that can apply rights to memory regions and exclude certain types of operations such as execution in the data space. A proper test on the target could be obtained. Merging with Zephyr did not give satisfactory results due to difficulties in identifying the location of the stack and heap at runtime. Other attack mechanisms were investigated such as CRAs, including *return2libc*and ROP. The proposed CFI countermeasure is difficult to implement but seems to be a global solution for most attacks. At the end of the time limit, three out of ten attacks could be thwarted by using cannaries as a compiler directive.

On a personal point of view, this project was the opportunity to learn more about hardware security and RISC-V ISA as well. Moreover, understanding and using the working environment with Zephyr RTOS booted on the CV32A6 was a real challenge. We had the chance to modify an OS and a core in SystemVerilog, which was a new language for both of us. In addition, we had to read articles and research reports, which is a rewarding exercise. The number of defeated attacks is lower than expected, partly because of the valuable practical results obtained with the PMP. Nevertheless, we are very glad to have participated in the competition.

ACKNOWLEDGMENT

Special thanks to Thales, GDR SOC² and the CNFM for organizing this contest and allowing students to work on exciting projects. It was a wonderful opportunity to learn more about a CPU architecture, the RISC-V ISA and hardware security in general. Many thanks to Maria Real Mendez who gave us the opportunity to work on this exciting project. Also, special thanks to Juliette Pottier for the time she took to help us.

REFERENCES

[1] Waterman, Andrew and Lee, Yunsup and Patterson, David and Asanovic, Krste, “The RISC-V instruction set manual, Volume I: User-Level ISA”, University of California at Berkeley Berkeley United States, 2014.

[2] Waterman, Andrew and Lee, Yunsup and Avizienis, Rimas and Patterson, David A and Asanovic, Krste, “The RISC-V instruction set manual, Volume II: Privileged architecture version 1.7”, University of California at Berkeley Berkeley United States, 2015.

[3] John Wilander and Nick Nikiforakis and Yves Younan and Mariam Kamkar and Wouter Joosen, “RIPE: Runtime Intrusion Prevention Evaluator”, In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC, 2011.

[4] John Wilander and Mariam Kamkar, “Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention”, Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA, The Internet Society, 2003.

[5] Baty, Matthieu and Hiet, Guillaume and Wilke, Pierre, “Work in progress: A formally verified shadow stack for RISC-V”

[6] Bulba and Kil3r, “Bypassing StackGuard and StackShield”, Phrack Magazine, Volume 10 Issue 56, 2000.

[7] Jaloyan, G. A., Markantonakis, K., Akram, R. N., Robin, D., Mayes, K., Naccache, D. (2020, October). “Return-oriented programming on RISC-V”. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (pp. 471-480).